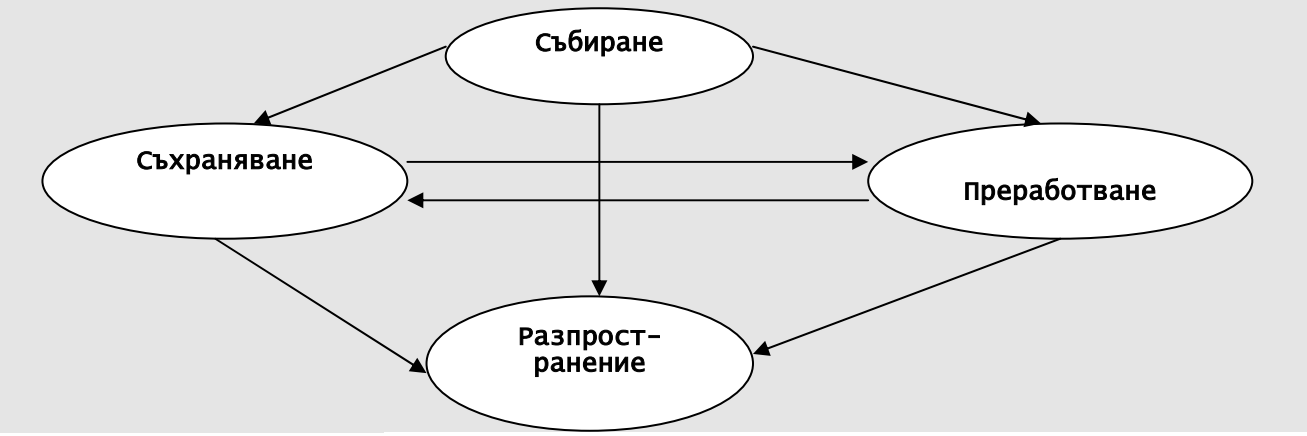
# 1

# Въведение в компютрите и програмирането

## 1.1 Компютър

Повечето от вас са използвали вече компютър за работа, за комуникации или за забавление. Компютърът е устройство или система, която е годна да изпълнява редица от операции по точно определен начин. Операциите са често числови изчисления, аналитични преобразувания или обработка на данни, но включват също и входно/изходни операции. Чрез компютрите се извършват: текстообработка, графика, счетоводство, управление на устройства, телекомуникационни и банкови услуги и др. дейности. Тъй като исторически първото им предназначение са изчисленията, наричат се още **изчислителни системи**.

Най-общо, компютърът е средство за представяне и обработване на информация. Неформално информацията е съвкупност от символи, над които се извършват следните информационни дейности: *събиране*, *съхраняване*, *преработване* и *разпространение*. Връзките между тези дейности са илюстрирани на Фиг. 1.1.

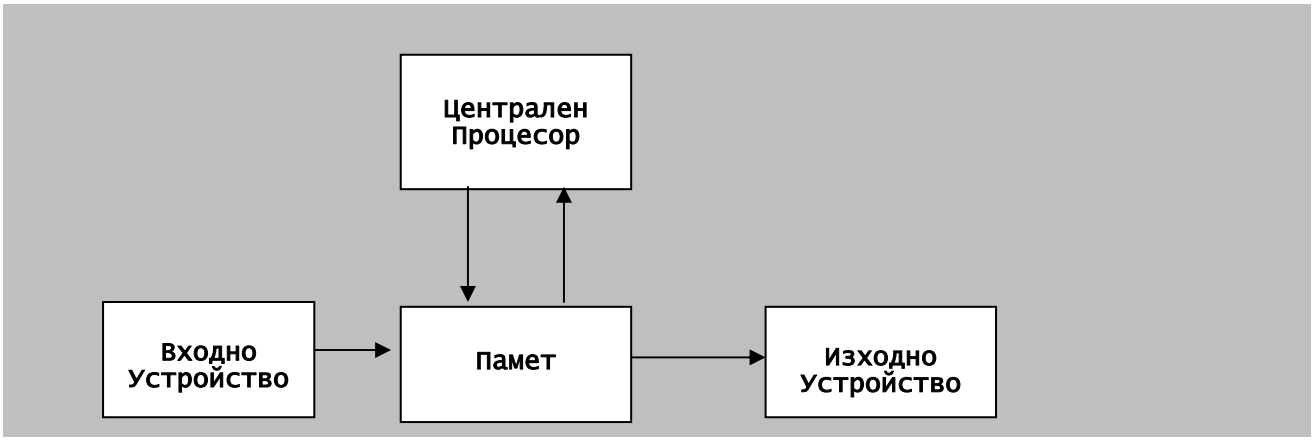



Фиг. 1.1 Връзки между информационните дейности

До сегашните компютри се стига след преминаване през следните поколения:

- *нулево поколение* – електронно механични устройства
- *първо поколение* – лампови ЕИМ - ENIAC (Electronic Numerical Integrator And Computer) е първия използваем компютър. Проектиран е от Дж. Преспър Екерт и Дж. Мокли от университета в Пенсилвания и е завършен през 1946 година. Компютърът е бил с огромни размери. Съдържал е около 18000 електрически лампи.
- *второ поколение* – транзисторни ЕИМ (1948)
- *трето поколение* – компютри на базата на интегрални схеми (миникомпютри)
- *четвърто поколение* – компютри на базата на големи интегрални схеми (микрокомпютри)

През 1946 година, на базата на основните информационни дейности и връзките между тях, Джон фон Нойман създава архитектура на изчислителна система, която преминава през горните поколения и има следния най-общ вид:

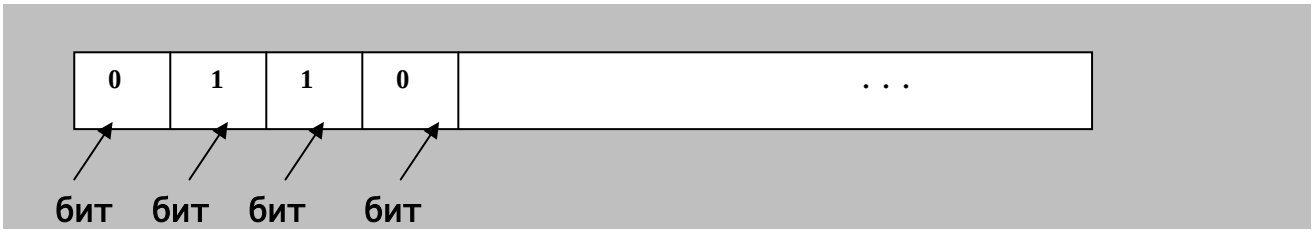



Фиг. 1.2. Архитектура на изчислителна машина

Информацията, която трябва да бъде обработена от компютъра, първо се въвежда в паметта му. Това се осъществява от входните му устройства. Централният процесор (ЦП) преработва информацията от паметта. Резултатът от тази обработка се запомня отново в паметта. Изходните устройства изобразяват информацията от паметта в подходящ (удобен) за потребителя вид.

*Памет*

Паметта на компютъра може да бъде изобразена като редица от елементи, всеки от които е носител на информацията – цифрите 0 и 1 (фиг. 1.3). Такъв обем информация се нарича **бит**.




Фиг. 1.3. Организация на паметта по битове

Технически не е възможно да бъде реализиран пряк достъп до всеки бит от паметта, поради което няколко бита се групират в **дума**. Думата може да е с размер 8, 16, 32, 64 и т.н. бита. 8 – битовата информационна единица се нарича **байт**. Обемът на паметта се измерва в байтове (B), килобайтове (KB), мегабайтове (MB), гигабайтове (GB) и терабайтове (TB), където:

1 KB = $2^{10}$ B ≈ $10^3$ B

1 MB = $2^{20}$ B ≈ $10^6$ B

1 GB = $2^{30}$ B ≈ $10^9$ B

1 TB = $2^{40}$ B ≈ $10^{12}$ B.

Обикновено се реализира пряк достъп до всяка дума на паметта. Всяка дума се свързва с пореден номер, наречен неин **адрес**. Номерацията започва от 0. На Фиг. 1.4. е изобразена памет на компютър, съдържаща 1000 думи, номерирани от 0 до 999.

Фиг. 1.4 Памет, съдържаща 1000 думи

В паметта на компютъра може да се представя всякакъв вид информация – числа, имена, списъци, картини. Информацията, която се записва в думите на паметта, се нарича **стойност на клетките на паметта**. Всяка дума на паметта съдържа някаква информация (няма празни клетки). Освен това, никоя дума не може да съдържа повече от една даннова единица. Всеки път когато някаква информация се записва в дума от паметта, старата информация в тази дума се унищожава и не може да бъде възстановена. Освен данни, в паметта на компютъра се записват и инструкции (команди), с помощта на които се обработват данните. Има два вида памет: **оперативна (ОП)** и **външна (ВП)**.

ОП е бърза, но скъпа. В съвременните компютри (микрокомпютрите) тя е изградена от чипове. **Компютърният чип** (интегрална схема) се състои от пластмасов или метален корпус, метални конектори и вътрешни връзки, изградени главно от силикон.

Има два типа ОП: **постоянна (ROM)** и **памет с произволен достъп (RAM)**. Паметта ROM съхранява информация, дори когато компютърът е изключен. Най-важните програми на микрокомпютъра са записани в паметта ROM, включително и програмата, която при включване на компютъра проверява дали всичките му компоненти работят правилно. За разлика от паметта ROM, RAM съхранява информация, само когато компютърът е включен. Нейният обем се измерва в килобайтове и мегабайтове.

Първият модел IBM PC има 256 KB RAM в системната си платка, но паметта му можеше да се разширява до 640 KB чрез включване на допълнителна платка памет към един от куплунгите за разширение на конфигурацията.

ВП осигурява по-евтино съхранение на информация, която се запазва при изключване на захранването. На тази памет информацията се съхранява във вид на файлове.

**Файлът** е организирана съвкупност от взаимно свързани данни. Изграден е от отделни части информация, наречени **записи**. Всеки файл има **име**, с което той е определен еднозначно в множеството от файлове.

Най-често за ВП се използва **твърд диск** (hard disk). Той се състои от въртящи се плочи с магнитно покритие и глави за четене и писане. Често се използва и **флопидиск** или **дискета**. Дискетата се състои от пластична кръгла основа, покрита с намагнитизирано вещество и

поставена в пластмасова обвивка. Използва се за пренасяне на информация от един компютър на друг. Дискетите са евтини, удобни и сравнително стабилни носители на информация. Основният им недостатък е, че имат малък обем. Широкоразпространени напоследък са компакт дисковете CD ROM. Те побират голямо количество информация, евтини са, но са предназначени само за четене.

*Централен процесор*

Той управлява цялостната работа на компютъра. Изграден е от един или няколко чипа. Вътрешността му е изключително сложна. Например, чипът на Pentium (известна марка процесори) е съставен от над 3,2 милиона транзистора. Произведен е чрез т. нар. 0,8 или 0,6 микронен фотолитографски процес. 0,8 и 0,6 означават широчината в микрони на металните пътечки, които свързват транзисторите му (1 $\mu$ = $10^{-6}$ m).

ЦП се грижи за управлението на системата, за аритметиката и прехвърлянето на данните. Той открива и изпълнява инструкции, извършва аритметични и логически операции, взема данни от паметта и външните устройства и ги връща обратно там.

ЦП се състои от управляващо усройство (УУ) и аритметично-логическо устройство (АЛУ).

УУ координира работата на компютъра. То изпраща разрешаващи и забраняващи сигнали към отделните устройства и по-такъв начин синхронизира работата на системата.

АЛУ изпълнява различни аритметични и логически операции (събиране, изваждане, умножение, деление, логическо събиране, логическо умножение, отрицание, сравнение и др.). Наборът от аритметични и логически операции определя т. нар. **език на компютъра** или **машинен език**. Всеки компютър може да изпълнява инструкциите на своя машинен език.

Първият модел IBM PC използва за ЦП микропроцесора Intel/8088. Този процесор може да обработва 16 бита (2 байта) информация едновременно. Тактовата честота на компютъра беше 4.77 MHz (мегахерца), което означава, че микропроцесорът получава 4.77 милиона импулса всяка секунда. Днес бързодействието на ЦП е многократно по-голямо от това на първия модел IBM PC. Технологичните подобрения на микропроцесорите се реализират с невероятни темпове.

Благодарение на това, производителността на микропроцесорите се удвоява на всеки 18 месеца.

*Входни и изходни устройства*

Наричат се още периферни устройства.

Входните устройства служат за въвеждане на информация в паметта на компютъра. Широкоразпространени са клавиатурата, мишката, дисковите и лентовите устройства, камерите, скенерите.

Изходните устройства извеждат резултатите в подходяща за възприемане форма. Най-разпространени са мониторите, дисковите и лентовите устройства, принтерът, тонколонките.

ЦП, RAM паметта и електрониката, която управлява твърдия диск и други устройства, са свързани помежду си посредством множество електрически линии, наречени **шина**. Данните и инструкциите се движат по шината от паметта и периферните устройства до ЦП и обратно. На дънната платка са поставени ЦП, RAM паметта и слотовете, чрез които платките, управляващи периферните устройства, се свързват с шината.

Съвкупността от електронните, електромеханичните и механичните компоненти, изграждащи компютъра, се нарича негова **апаратна част** или **хардуер**. Хардуерът на компютъра е тясно свързан с т. нар. **програмна част**, **програмно осигуряване** или **софтуер**.

Програмното осигуряване на компютрите включва:

- операционни системи;
- среди за програмиране;
- приложни програми.

*Операционни системи (ОС)*

Всеки компютър работи под управлението на някаква ОС. Тя управлява:

- взаимодействието между потребителите и хардуера (чрез команден език);
- периферните устройства (чрез драйвери);
- файловете (чрез файловата система).

Широкоразпространени ОС са MS DOS, UNIX, LINUX.

*Среди за програмиране (СП)*

Те автоматизират цялостния процес по създаването, транслирането, тестването, изпълнението, изменението и документирането на

програмите. Средите за програмиране включват: *език за програмиране, транслатор (компилатор или интерпретатор), свързващ редактор, изпълнителна система, система за проверка на програми, система за поддържане на библиотеки и текстови редактори*. Тези компоненти са свързани в система чрез управляваща програма.

В настоящия курс по програмиране ще използваме средата за програмиране Visual C++ 6.0.

*Приложни програми (ПП)*

Те реализират някакво приложение – икономическо, проектантско, инженерно, счетоводно, медицинско и др.

## 1.2. Програми и програмиране. Езици за програмиране

### 1.2.1. Програми. Автоматизиране на програмирането

За да бъде изпълнена една или друга обработка над входните данни, трябва да бъде написана редица от инструкции (команди), които компютърът разбира и изпълнението на които да води до решаването на някаква задача. За различните задачи, тези редици са различни.

Редица от инструкции, водеща до решаване на определена задача, се нарича **програма**, а процесът на писане на програми – **програмиране**.

Инструкциите, с помощта на които се записва програма, изграждат език, наречен **език за програмиране**. Всеки компютър си има свой собствен език за програмиране (машинен език) и може да изпълнява програми, написани на него.

Машинният език е множество от машинни инструкции, които процесорът на компютъра може директно да изпълни. Програмата на машинен език е редица от числа, записани в двоична позиционна система.

Една типична редица от машинни инструкции е следната:

1. Премести стойността на клетка 15000 от паметта в регистър AX.
2. Извади 10 от регистъра AX.
3. Ако резултатът е положително число, изпълни командата, намираща се в клетка 25000 от паметта.

За различните процесори тази редица от инструкции се кодира по различен начин. При процесора Intel 80386 това кодиране има вида:

161 15000 45 10 127 25000

Тази редица от числа се записва в паметта, след което се изпълнява. Но една дълга програма е съставена от хиляди команди и вероятността за грешки при програмиране на машинен език е много голяма.

Първата стъпка в процеса на автоматизация на програмирането е въвеждането на **асемблерните езици**. Те дават кратки имена на командите. Например MOV означава премести, SUB – извади, а JG – ако е по-голямо от 0, премини към …. Като се използват тези команди, горната редица от инструкции се свежда до:

```
MOV AX, [15000]
SUB AX, 10
JG 25000
```

Този фрагмент е по-лесен за четене от хора, но компютърът не разбира тези инструкции. Те трябва да бъдат преведени на машинен език. Тази задача се изпълнява от компютърна програма, наречена **асемблер**. Асемблерът взема редицата от букви MOV AX и я превежда в машинния код 161. Подобни действия асемблерът извършва и над другите команди. Асемблерът именува също и думите от паметта. Това позволява да бъдат използвани буквени имена вместо адреси от паметта. Например, горният фрагмент може да се запише и във вида:

```
MOV AX, [A]
SUB AX, 10
JG WRITE_ERROR
```

и извършва следните действия: Прехвърля стойността на променливата A в регистъра AX; Изважда 10 от регистъра AX; Ако стойността на полученото е положително число, извежда съобщение за грешка.

Асемблерните езици имат големи предимства пред използването на чисти машинни кодове, но имат и съществени недостатъци. Двата най-важни техни недостатъка са, че първо, дори и за най-простите задачи е необходим голям брой инструкции и, второ, редицата от команди се променя от процесор на процесор. Ако компютърът излезе от употреба, програмите трябва да се напишат отново за новата машина.

В средата на 50-те години започнаха да се появяват езици за програмиране от високо ниво. При тях програмистът описва схематично основната идея за решаване на задачата, а специална програма, наречена **транслатор (компилатор** или **интерпретатор)**, превежда това описание в машинни инструкции за конкретния процесор. Например, на езика Паскал, горната редица от инструкции ще се запише във вида:

```
if a > 10 then writeln('error')
```

Работата на компилатора е да обработи тази редица от букви и да я преведе във вида:

161 15000 45 10 127 25000

Транслаторите са доста сложни програми. В рамките на настоящия курс ще ги приемем за даденост. Важното е, че езиците за програмиране от високо ниво са независими от хардуера.

## 1.2.2. Езици за програмиране

### 1.2.2.1. Класификация

Сега съществуват стотици различни езици за програмиране. Те могат да бъдат класифицирани по различни признаци. Най-общоприетата класификация е по степента на зависимост на езика от компютъра. Фиг. 1.5 илюстрира тази класификация.

| машинно-зависими | | | машинно-независими | |
|---|---|---|---|---|
| машинни | асемблерни | макроезици | процедурно ориентирани | проблемно ориентирани |

Фиг. 1.5 Класификация на езиците за програмиране

Машинно-независимите езици се наричат още **езици от високо ниво.**

В настоящия курс ще направим въведение в програмирането на базата на езика за процедурно и обектно-ориентирано програмиране C++, който притежава и средства за програмиране почти на машинно ниво.

Друга класификация на езиците за програмиране е *според стила на програмиране.* Съществуват два основни стила за програмиране: **императивен (процедурен)** и **декларативен (дескриптивен).**

При процедурния стил програмата се реализира по схемата:

**Програма = Алгоритъм**

**+**

**Структури от данни**

Съществуват стотици процедурни езици. Такива са машинните и асемблерните езици, езиците Фортран, Алгол, Кобол, Снобол, Паскал, Ада, С, С++ и др.

Процедурният стил е тясно свързан с фон Нойманова архитектура на изчислителна система. Процедурните езици за програмиране могат да се разглеждат като синтактични обобщения на Нойманови компютри. Семантиката на тези езици по същество е същата като семантиката на машинните езици. Следователно, процедурните езици са абстрактни версии от високо ниво на фон Нойманови компютри и като такива, те наследяват всички техни недостатъци.

Един от учените, на който до голяма степен се дължи увековечаването на архитектурата на фон Нойман и на процедурното програмиране е професорът от IBM Дж. Бекус. Той е авторът на езика Фортран, ръководил е редица проекти, свързани с реализиране на процедурни езици за програмиране. Но Бекус, анализирайки недостатъците на фон Нойманова архитектура, през 1978 година на конгрес на IFIP повдига въпроса: *Може ли програмирането да се освободи от бремето на Нойманови стил?* Да разгледаме какви са основните недостатъци на императивните езици?

Линията за връзка между ОП и ЦП е тясното място на Нойманови компютри. Бекус я нарича “гърло на бутилката”. Изпълнението на машинната програма води до съществено изменение на ОП. Това се достига чрез многократно прехвърляне на стойността на отделни клетки в едната и другата посока. При това обаче съществена част от трафика през това място не са “полезни” данни, а адреси на данни, а също операции, които се използват за пресмятане на такива адреси. Бекус отбелязва, че тясното място на компютъра създава и тясно място в мисленето на програмиста като го кара да си представя цялата обработка на данните клетка по клетка, вместо да решава задачата си в по-мащабни термини. Програмата, написана на императивен език, работи с клетките на паметта, отделени за променливите. Програмистът трябва да опише всички последователни изменения на тази памет, така че след като те се изпълнят, да се получи търсеният резултат. Така императивната програма трябва да управлява последователните стъпки на работа на компютъра. От необходимостта, обработката на данните в стил “клетка след клетка” да се опише точно и детайлно, следват следните недостатъци: *относително малка мощност, липса на гъвкавост, ниска производителност*.

Важен момент при създаването на езици за програмиране е формалното описание на семантиката на езика. Би било добре това описание да е просто и удобно за доказване на редица свойства на програмите (частична и тотална коректност, завършване на изпълнението, еквивалентност на програмите и др.). Последното не е така при императивните езици, тъй като те са сложни (косвено следствие от Нойманова архитектура). Оттук пък произтича твърде голямото и сложно формално описание на семантиките им. Поради сложността си, формалното описание на семантиката на императивен език за програмиране не е много по-полезно от неформалното описание на езика. На практика то се използва само при изследването свойствата на малки и елементарни програми, но не и при по-сложни реални задачи. *Така вторият главен недостатък на императивните езици е липсата на полезни математически свойства. Чрез императивните езици не могат практически лесно да се доказват свойства на програмите.*

Все пак връзката на императивните езици с архитектурата на фон Нойман има и предимства. Главното предимство е ефективната им реализация.

И Бекус, анализирайки недостатъците на императивните езици предлага нов стил за програмиране – FP стила (вид декларативно програмиране).

Декларативните езици са антипод на императивните. При тях в програмата се указва явно какви свойства има желаният резултат, без да се указва как точно този резултат се получава.

Допустим е всеки начин за построение на програмата, който води до получаването на резултат с описаните в програмата свойства. Така при декларативните езици се казва какво се пресмята, без да се определя как да се пресмята. Програмата на декларативен език за програмиране се определя така:

**Програма = Дефиниции_на_функции или**
**Равенства или**
**Правила + Факти**

а изпълнението й е **редукция, извод** или **доказателство.**

Такива езици са Лисп, Пролог, Pop-11, TabLog, EqLog, Prolog-with-Equality и др.

Тези езици **не** са непосредствено свързани с фон Нойманова архитектура. От това произтича един съществен техен недостатък – ниска ефективност на реализацията им на такива компютри. Така възниква необходимостта от т. нар. ненойманови компютри или *компютри от пето поколение*. Това са компютри, които ще се характеризират с висока степен на паралелизация.

Сегашните прогнози за тези компютри са свързани с технологията на 21-ви век, определена като нанотехнология или молекулярно производство. При това производство, измерванията ще са в нанометри ($1\eta = 10^{-9}m$), а не в микрони ($1\mu = 10^{-6}m$), както е в момента. Представете си стотици хиляди микропроцесора, работещи паралелно, събрани в блокче с размери на бучка захар, като всички принадлежности на персонален компютър се побират в този обем. Предвижда се молекулярното производство да доведе до прецизно управление на производствените процеси на широк кръг механични устройства.

В момента не една технология произвежда все по-малки и по-малки компоненти. Фирмата Hitachi демонстрира запаметяващо устройство само от един електрон, съхраняващ 1 бит информация. Texas Instruments са произвели чип с компоненти 50 пъти по-малки от обикновените. Toshiba е изработила транзистор с широчина 0.04 микрона.

Китайският професор Гао Хунджун, използвайки органични молекулни материали, е създал електрическо поле в молекула, която представлява основна единица за съхраняване на данни. Дължината на молекулата е около 1 нанометър и е основната мярка на информационния капацитет. На базата на това откритие е създаден записващ процес, позволяващ на устройство с размерите на CD да се запишат 1 милион пъти повече информация, отколкото в компакт диск. Това е твърде привлекателно изследване, което може да се използва за създадане на памет, основана на органична материя. Вероятно ще минат 10-15 години преди да бъде внедрено това изследване. Основният му недостатък е, че органичният материал, използван сега, е нестабилен – губи паметта си за около месец.

#### 1.2.2.2. Описание на език за програмиране

За да се опише един език за програмиране е необходимо да се опишат неговите **синтаксис** и **семантика**. Всяка програма на някакъв

език за програмиране може да се разглежда като редица от символи. Но не всяка редица от символи е програма на съответния език. Да се определи **синтаксисът на един език за програмиране** означава да се зададе множество от правила, които определят как програмите се изграждат като редици от символи, т.е. да се зададе множество от правила, съгласно които една редица от символи е програма. Да се определи **семантиката на език за програмиране** означава да се определят правилата, съгласно които се изпълняват програмите на този език за програмиране. Следователно, синтаксисът определя формата, а семантиката – смисъла на програмата. Задаването става словестно, чрез примери или чрез правила.

При изучаването на езика C++ синтаксисът му ще описваме словестно и чрез метаезика на Бекус-Наур, а за описание на семантиката му ще използваме словестното описание.

*Метаезик на Бекус-Наур*

Това е първото широкоизползвано формално означение за описание на синтаксиса на език за програмиране. Въведено е като механизъм за описание на синтаксиса на езика Algol 60.

При това описание, имената на синтактичните категории се ограждат чрез ъглови скобки.

*Пример*

<програма>, <число>, <цяло_число>

са правилно означени синтактични категории.

Знакът ::= означава “това е” или “дефинира се да бъде”. Синтактичните категории, които се дефинират, се поставят отляво на знака ::=, а дефиницията им – отдясно на този знак.

*Пример*

Означението

<реално_число> ::= <цяло_число>.<цяло_число_без_знак>

определя, че реално число е конструкция, започваща с цяло число, следвано от символа точка и от цяло число без знак.

Знакът | определя алтернативни конструкции и означава ИЛИ.

*Пример*

Описанието

<цифра> ::= 0|1|2| … |9

означава, че цифра се дефинира като 0 или 1, или 2, или и т.н., или 9.

Означението {…} има смисъла, че ограденото в него се повтаря 0, 1 или повече пъти.

*Пример*

<редица_от_променливи> ::= <променлива> {, <променлива>}

означава, че редица от променливи се дефинира като

<променлива> или

<променлива>, <променлива> или

<променлива>, <променлива>, <променлива> и т.н.

Означението […] има смисъла, че ограденото в скобите може да участва в описанието, но може и да бъде пропуснато.

*Пример*

Описанието

<цяло_число> ::= [+|_]<цяло_число_без_знак>

означава, че цяло число се дефинира или като цяло число без знак, или като цяло число, предшествано от знак + или -.

## 1.3. Алгоритъм

При процедурния стил програмата се реализира по схемата:

**Програма = Алгоритъм**

**+**

**Структури от данни**

Основен неин елемент е алгоритъмът.

Механизъм за намиране на решение, който е еднозначен, изпълним и завършващ, се нарича **алгоритъм**.

Алгоритъмът като понятие е въведено от Евклид. Наречен е така в чест на арабския математик Ал Хорезми. В средновековието формулата DIXIT ALGORIZMI (така е казал Ал Хорезми) била сертификат за безупречна яснота и достоверност.

*Еднозначността* гарантира точността на инструкциите на всяка стъпка и определя следващото действие. Свойството *изпълнимост* определя, че всяка стъпка може да се изпълни на практика, а *завършването* – че изпълнението на стъпките ще завърши.

Алгоритъмът се описва чрез редица от правила, които имат три компоненти: *пореден номер, команда* и *наследник*. Наследникът може да се пропусне ако съвпада със следващата команда от редицата.

Основни начини за описние на алгоритъма са словестният, чрез блок-схема, чрез средствата на някакъв език за програмиране.

Ще илюстрираме словестното описние чрез пример.

*Пример.* Следващият фрагмент дава *общо* описание на алгоритъм за намиране на най-големия елемент на редицата от числа $a_1$, $a_2$, …, $a_n$.

1. $m = a_1$
2. Сравнява се $a_2$ с m и ако $a_2$ е по-голямо от m , то $m = a_2$
3. Стъпка 2 се повтаря до изчерпване на редицата.

Следва детайлно описание на алгоритъма за намиране на най-големия елемент на редицата от числа $a_1$, $a_2$, …, $a_n$.

1. $m := a_1$
2. $i := 2$
3. **Ако** $a_i > m$ **То** изпълни 4 **ИНАЧЕ** изпълни 5
4. $m := a_i$
5. $i := i + 1$
6. **Ако** $i \leq n$ **То** изпълни 3 **Иначе** изпълни 7
7. max := m
8. съобщи max
9. прекрати изпълнението

При това описание n и $a_1$, $a_2$, …, $a_n$ съдържат входните данни и се наричат още **входни променливи**, max съдържа резултата и се нарича **изходна променлива**, а m и i са **работни (вътрешни, междинни, помощни) променливи**.

Решаването на една задача чрез програма на процедурен език се свежда до точно описание на това как да се получи резултатът.

## Допълнителна литература


1. П. Азълов, Програмиране. Основен курс. Увод в програмирането, София, АСИО, 1995.
2. К. Хорстман, Принципи на програмирането със C++, София, СОФТЕХ, 2000.

# 2

# Основни елементи от програмирането на C++

C++ е език за обектно-ориентирано програмиране. Създаден е от Бярне Страуструп от AT&T Bell Laboratories в края на 1985 година. C++ е разширение на езика C в следните три направления:

- създаване и използване на абстрактни типове данни;
- обектно-ориентирано програмиране;
- подобрения на конструкции на езика C (производни типове, наследяване, полиморфизъм).

През първите шест месеца след описанието му се появиха над 20 търговски реализации на езика, предназначени за различни компютърни системи. От тогава до сега C++ се разраства чрез добавяне на много нови функции и затова процесът на стандартизацията му продължава и до момента. C++ е пример за език, който с времето расте и се развива. Всеки път, когато потребителите му са забелязвали някакви пропуски или недостатъци, те са го обогатявали със съответните нови възможности.

За разлика от C++, езикът Паскал е създаден планомерно главно за целите на обучението. Проф. Вирт добре е проектирал и доказал езика. Тъй като Паскал е създаден с ясна цел, отделните му компоненти са логически свързани и лесно могат да бъдат комбинирани. Разрастващите се езици, към които принадлежи C++ са доста обърканн тъй като хора с различни вкусове правят различни нововъведения. Освен това, заради мобилността на програмите, не е възможно премахването на стари конструкции, даже да съществуват удобни техни подобрения. Така разрастващият се C++ събира в себе си голям брой възможности, които не винаги добре се съвместяват.

Езиците, създадени от компетентни хора, по принцип са лесни за научаване и използване. Разрастващите се езици обаче държат монопола на пазара. Сега C++ е водещия език за програмиране с общо

предназначение. Лошото е, че не е много лесен за усвояване, има си своите неудобства и капани. Но той има и огромни приложения – от програми на ниско, почти машинно ниво, до програми от най-висока степен на абстракция.

Целта на настоящия курс по програмиране е не да ви научи на всички възможности на C++, а на изкуството и науката програмиране.

При началното запознаване с езика, възникват два естествени въпроса:

- *Какво е програма на C++ и как се пише тя?*
- *Как се изпълнява програма на C++?*

Ще отговорим на тези въпроси чрез пример за програма на C++, след което ще дадем някои дефиниции и основни означения.

## 2.1. Пример за програма на C++

**Задача 1.** Да се напише програма, която намира периметъра и лицето на правоъгълник със страни 2,3 и 3,7.

Една програма,която решава задачата е следната:

```
// Program Zad1.cpp
#include <iostream.h>
int main()
{double a = 2.3;
  double b = 3.7;
  double p, s;
  /* намиране на периметъра
     на правоъгълника */
  p = 2*(a+b);
  /* намиране на лицето на правоъгълника */
  s = a*b;
 /* извеждане на периметъра */
  cout << "p= " << p << "\n";
  /* извеждане на лицето */
  cout << "s= " << s << "\n";
  return 0;
}
```

Първият ред

```
// Program Zad1.cpp
```

е коментар в езика C++. Започва с // и завършва в края на реда (ENTER). След знаците // е записан текст, предназначен за програмиста и подсещащ за смисъла на следващото действие. В случая коментарът информира, че следващият фрагмент е програма на езика C++ с име Zad1.cpp и по-точно, че следващата програма е записана във файл с име Zad1.cpp.

Линията

```
#include <iostream.h>
```

е **директива към компилатора на C++**. Чрез нея към файла Zad1.cpp, съдържащ програмата, се включва файлът с име iostream.h (При някои реализации на C++ разширението ".h" се пропуска). Този файл съдържа различни дефиниции и декларации, необходими за реализациите на операциите за поточен вход и изход. В програма Zad1.cpp се нуждаем от тази директива заради извеждането върху екрана на периметъра и лицето на правоъгълника.

Конструкцията

```
int main()
{ …
   return 0;
}
```

дефинира **функция**, наречена **main** (главна). Всяка програма на C++ трябва да има функция main. Повечето програми съдържат и други функции освен нея.

Дефиницията на main започва с **думата** int (съкращение от integer), показваща, че main връща цяло число, а не дроб или низ, например. Между фигурните скоби { и } е записана редица от **дефиниции** и **изпълними изрази** (**оператори**), която се нарича **тяло на функцията**. Компонентите на тялото се отделят със знака ; и се изпълняват последователно. С оператора return се означава краят на функцията. Стойността 0 означава, че тя се е изпълнила успешно. Ако програмата завърши изпълнението си и върне стойност различна от 0, това означава, че е възникнала грешка.

Конструкциите

```
double a = 2.3;
double b = 3.7;
double p, s;
```

дефинират **променливите** a, b, p и s от реалния тип double, като в първите два случая се дават начални стойности на a и b (2.3 и 3.7

съответно). Казва се още, че **a и b са инициализирани с 2.3 и 3.7 съответно.**

Променливата е място за съхранение на данни, което може да съдържа различни стойности по време на изпълнение на програмата. Означава се чрез редица от букви, цифри и долна черта, започваща с буква или долна черта. Променливите имат три характеристики: **тип**, **име** и **стойност**. Преди да бъдат използвани, трябва да бъдат дефинирани.

C++ е строго типизиран език за програмиране. Всяка променлива има тип, който *явно* се указва при дефинирането й. Пропускането на типа на променливата води до сериозни грешки. Фиг. 2.1. илюстрира непълно дефинирането на променливи.

**Дефиниране на променливи**

*Синтаксис*

<име_на_тип> <променлива> [= <израз>]$_{опц}$
{, <променлива> [= <израз>]$_{опц}$}$_{опц}$;

където

<име_на_тип> е дума, означаваща име на тип като int, double и др.;

<израз> е правило за получаване на стойност – цяла, реална, знакова и друг тип, съвместим с <име_на_тип>.

*Семантика*

Дефиницията свързва променливата с множеството от допустимите стойности на типа, от който е променливата или с конкретна стойност от това множество. За целта се отделя определено количество оперативна памет (толкова, колкото да се запише най-голямата константа от множеството от допустимите стойности на съответния тип) и се именува с името на променливата. Тази памет е с неопределена стойност или съдържа стойността на указания израз, ако е направена инициализация.

*Пример:*

```
double a = 2.3;
double b, p, s;
```

Не се допуска една и съща променлива да има няколко дефиниции в рамките на една и съща функция.

Фиг. 2.1 Дефиниране на променливи

*Забележка:* При описанието на синтаксиса, означенията […] и {…} от езика на Бекус-Наур, ще завършваме с индекса опц.

В случая на програмата Zad1.cpp за a, b, p и s се отделят по 8 байта оперативна памет и

ОП

| a | b | p | s |
|---|---|---|---|
| 2.3 | 3.7 | - | - |
| 8 байта | 8 байта | 8 байта | 8 байта |

Неопределеността на p и s е означена с -.

След дефинициите на променливите a, b, p и s е разположен коментарът

/* намиране на периметъра на правоъгълника */

Той е предназначен за програмиста и подсеща за смисъла на следващото действие.

Коментарът (Фиг. 2.2) е произволен текст, ограден със знаците /* и */ или от // и ENTER. Игнорира се напълно от компилатора. Ще напомним, че средите за програмиране включват специална програма, наречена *транслатор (компилатор или интерпретатор),* която превежда програмата в програма, записана на машинен език. В средата Visual C++ 6.0 транслаторът е реализиран като компилатор.

**Коментар**

*Синтаксис*

```
<коментар> ::= /* <редица_от_знаци> */ |
               // <редица_от_знаци> ENTER
```

*Семантика*

Пояснява програмен фрагмент. Предназначен е за програмиста. Игнорира се от компилатора на езика.

*Примери:*

```
/* Намиране на лицето
   и периметъра на правоъгълник */
// Program Zad1.cpp
```

Фиг. 2.2 Коментар

Конструкциите

```
p = 2*(a+b);
s = a*b;
```

са **оператори за присвояване на стойност** (Фиг. 2.3). Чрез тях променливите p и s получават текущи стойности. Операторът

```
p = 2*(a+b);
```

пресмята стойността на аритметичния израз 2*(a+b) и записва полученото реално число (в случая 12.0) в паметта, именувана с p. Аналогично, операторът

```
s = a*b;
```

пресмята стойността на аритметичния израз a*b и записва полученото реално число (в случая 8.51) в паметта, именувана със s.

По-подробно ще разгледаме този оператор в следващата глава. На този етап оставяме с интуитивната представа за <израз>.

**Оператор за присвояване**

*Синтаксис*

<променлива> = <израз>;

като <променлива> и <израз> са от един и същ тип.

*Семантика*

Пресмята стойността на <израз> и я записва в паметта, именувана с променливата от лявата страна на знака за присвояване =.

*Пример:*

```
p = 2*(a+b);
```

Фиг. 2.3 Оператор за присвояване на стойност

Да се върнем към дефинициите на променливите a и b и операторите за присвояване и return на Zad1.cpp. Забелязваме, че в тях са използвани два вида числа: **цели** (2 и 0) и **реални** (2.3 и 3.7). Целите числа се записват като в математиката, а при реалните, знакът запетая се заменя с точка. Умножението е отбелязано със *, а събирането – с +. Забелязваме също, че изразите 2*(a+b) и a*b са реални, каквито са и променливите p и s от левите страни на знака = в операторите за присвояване.

Конструкциите

```
cout << "p= " << p << "\n";
cout << "s= " << s << "\n";
```

са оператори за извеждане. Наричат се още оператори за поточен изход. Еквивалентни са на редицата от оператори:

```
cout << "p= ";
```

```
cout << p;
cout << "\n";
cout << "s= ";
cout << s;
cout << "\n";
```

Операторът **<<** означава “изпрати към”. Обектът (променливата) cout (произнася се “си-аут”) е името на стандартния изходен поток, който обикновено е екрана или прозорец на екрана.

Редица от знаци, оградена в кавички, се нарича **знаков низ**, или **символен низ**, или само **низ**. В програмата Zad1.cpp “p= “ и “s= “ са низове. Низът “\n“ съдържа двойката знаци \ (backslash) и n, но те представляват един-единствен знак, който се нарича **знак за нов ред**. Операторът

```
cout << "p= ";
```

извежда върху екрана низа p=

Операторът

```
cout << p;
```

извежда върху екрана стойността на p, а

```
cout << "\n";
```

премества курсора на следващия ред на екрана, т.е. указва следващото извеждане да бъде на нов ред.

Фиг. 2.7. показва по-детайлно синтаксиса и семантиката на оператора за извеждане.

*Изпълнение на Zad1.cpp*

След обработката на директивата

```
#include <iostream.h>
```

файлът iostream.h е включен във файла, съдържащ функцията main на Zad1.cpp. Изпълнението на тялото на main започва с изпълнение на дефинициите

```
double a = 2.3;
double b = 3.7;
double p, s;
```

в резултат, на което в ОП се отделят по 8 байта за променливите a, b, p и s, т.е.

ОП

| a | b | p | s |
|---|---|---|---|
| 2.3 | 3.7 | - | - |

Коментарите се пропускат. След изпълнението на операторите за присвояване:

```
p = 2*(a+b);
s = a*b;
```

променливите p и s се свързват с 12.0 и 8.51 съответно, т.е.

| ОП | | | |
|---|---|---|---|
| a | b | p | s |
| 2.3 | 3.7 | 12.0 | 8.51 |

Операторите

```
cout << "p= " << p << "\n";
cout << "s= " << s << "\n";
```

извеждат върху екрана на монитора

```
p= 12
s= 8.51
```

Изпълнението на оператора

```
return 0;
```

преустановява работата на програмата сигнализирайки, че тя е завършила успешно.

**Забележка:** Реалните числа се извеждат с възможно минималния брой знаци. Така реалното число 12.0 се извежда като 12.

В същност, описаните действия се извършват над машинния еквивалент на програмата Zad1.cpp. А как се достига до него?

**Изпълнение на програма на езика C++**

За целта се използва някаква среда за програмиране на C++. Ние ще използваме Visual C++ версия 6.0.

Изпълнението се осъществява чрез преминаване през следните стъпки:

*1. Създаване на изходен код*

Чрез текстовия редактор на средата, текстът на програмата се записва във файл. Неговото име се състои от две части – име и разширение. Разширението подсказва предназначението на файла. Различно е за отделните реализации на езика. Често срещано разширение за изходни файлове е “.cpp” или “.c”.

Примерната програма е записана във файла Zad1.cpp.

*2. Компилиране*

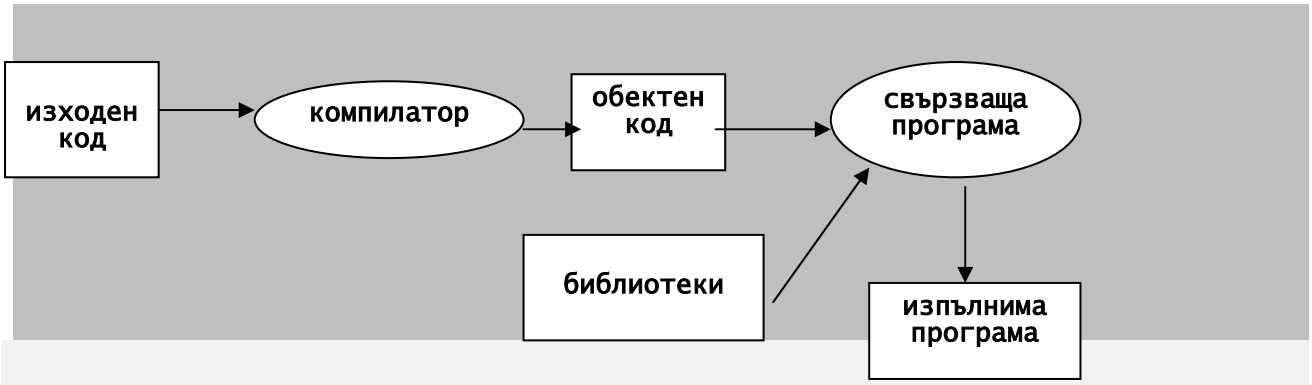
Тази стъпка се изпълнява от компилатора на езика. Първата част от работата на компилатора на C++ е откриването на грешки – синтактични и грешки, свързани с типа на данните. Съобщението за

грешка съдържа номера на реда, където е открита грешка и кратко описание на предполагаемата причина за нея. Добре е грешките да се коригират в последователността, в която са обявени, защото една грешка може да доведе до т. нар. “каскаден ефект”, при който компилаторът открива повече грешки, отколкото реално съществуват. Коригираният текст на програмата трябва да се компилира отново. Втората част от работата на компилатора е превеждане (транслиране) на изходния (source) код на програмата в т. нар. **обектен (object) код**. Обектният код се състои от машинни инструкции и информация за това, как да се зареди програмата в ОП, преди да започне изпълнението й. Обектният код се записва в отделен файл, обикновено със старото име, но с разширение “.obj” или “.o”.

Обектният файл съдържа само “превода” на програмата, а не и на библиотеките, които са декларирани в нея (в случая на програмата Zad1.cpp файлът Zad1.obj не съдържа обектния код на iostream.h). Авторите на пакета iostream.h са описали всички необходими действия и са записали нужния машинен код в библиотеката iostream.h.

*3. Свързване*

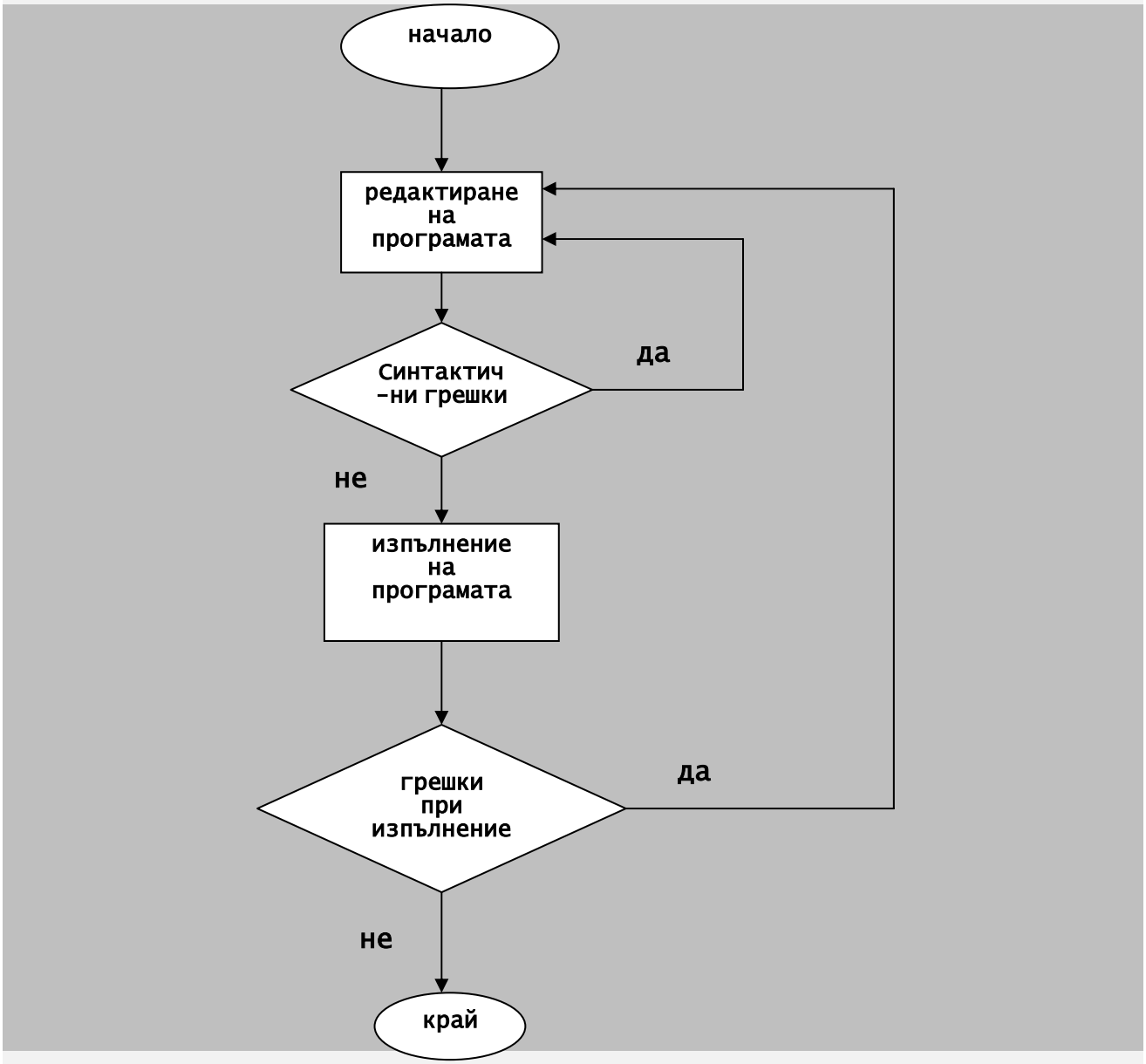
Обектният файл и необходимите части от библиотеки се свързват в т. нар. **изпълним файл**. Това се извършва от специална програма, наречена **свързваща програма** или **свързващ редактор** (**linker**). Изпълнимият файл има името на изходния файл, но разширението му обикновено е “.exe”. Той съдържа целия машинен код, необходим за изпълнението на програмата. Този файл може да се изпълни и извън средата за програмиране на езика C++. Фиг. 2.4 илюстрира стъпките на изпълнение на програма на C++.




Фиг. 2.4 Изпълнение на C++ програма

Програмистката дейност, свързана с изпълнението на програма на C++, преминава през три стъпки като реализира цикъла “редактиране-

компилиране-настройка”. Започва се с редактора като се пише изходният файл. Компилира се програмата и ако има синтактични грешки, чрез редактора се поправят грешките. Когато програмата е “изчистена” от синтактичните грешки, започва изпълнението й. Ако възникнат грешки по време на изпълнението, осъществява се връщане отново в редактора и се поправят предполагаемите грешки. После пак се компилира и стартира програмата. Цикълът “редактиране-компилиране-настройка” е илюстриран на Фиг. 2.5.




Фиг. 2.5 Цикъл редактиране-компилиране-настройка

## 2.2. Основни означения

Всяка програма на C++ е записана като редица от знаци, които принадлежат на **азбуката на езика**.

**Азбука на C++**

Азбуката на езика включва:

- главните и малки букви на латинската азбука;
- цифрите;
- специалните символи

  + - * / = ( ) [ ] { } | : ; “ ‘ < > , . _

  ! @ # $ % ^ ~

Някой от тези знаци, по определени правила, са групирани в думи (лексеми) на езика.

**Думи на езика**

Думите на езика са идентификатори, запазени и стандартни думи, константи, оператори и препинателни знаци.

*Идентификатори*

Редица от букви, цифри и знака за подчертаване (долна черта), започваща с буква или знака за подчертаване, се нарича **идентификатор**.

*Примери:*

Редиците от знаци

A12 help Help double int15_12 rat_number INT1213 Int15_12

са идентификатори, а редиците

1ba ab+1 a(1) a’b

не са идентификатори. В първия случай редицата започва със цифра, а в останалите – редиците съдържат недопустими за идентификатор знаци.

Идентификаторите могат да са с произволна дължина. В съвременните компилатори максималният брой знаци на идентификаторите може да се задава, като подразбиращата се стойност е 32.

*Забележка*: При идентификаторите **се прави разлика** между малки и главни букви, така help, Help, HELP, HeLp и HElp са различни идентификатори.

Идентификаторите се използват за означаване на имена на променливи, константи, типове, функции, класове, обекти и други компоненти на програмите.

*Препоръка:* Не започвайте вашите идентификатори със знака за подчертаване. Такива идентификатори се използват от компилатора на C++ за вътрешно предназначение.

*Допълнение:* Чрез метаезика на Бекус-Наур, синтаксисът на променливите се определя по следния начин:

<променлива> ::= <идентификатор>

Някои идентификатори са резервирани в езика.

*Запазени думи*

Това са такива идентификатори, които се използват в програмите по стандартен, по предварително определен начин и които не могат да бъдат използвани по друг начин. Чрез тях се означават декларации, дефиниции, оператори, модификатори и други конструкции. Реализацията Visual C++ 6.0 съдържа около 70 такива думи.

В програмата Zad1.cpp са използвани запазените думи int, double, return.

*Стандартни думи*

Това са такива идентификатори, които се използват в програмите по стандартен, по предварително определен начин. Тези идентификатори **могат** да се използват и по други начини, например като обикновени идентификатори.

В програмата Zad1.cpp е използвана стандартната дума cout.

Например,

```
#include <iostream.h>
int main()
```

```
{int cout = 21;
 return 0;
}
```

е допустима програма на C++. В нея идентификаторът cout е използван като име на променлива. Правенето на опит за използване на cout по стандартния начин води до грешка. Така фрагментът

```
#include <iostream.h>
  int main()
  {int cout = 21;
   cout << cout << "\n";
   return 0;
   }
```

е недопустим.

*Препоръка*: Стандартните думи да се използват само по стандартния начин.

*Константи*

Информационна единица, която не може да бъде променяна, се нарича **константа**. Има числови, знакови, низови и др. типове константи.

Целите и реалните числа са **числови константи**. Целите числа се записват както в математиката и могат да бъдат задавани в десетична, шестнадесетична или осмична бройна система. Реалните числа се записват по два начина: във формат с *фиксирана точка* (например, 2.34 -12345.098) и в *експоненциален формат* (например, 5.23e-3 или 5.23E-3 означават 5.23 умножено с $10^{-3}$).

**Низ**, **знаков низ** или **символен низ** е крайна редица от знаци, оградени в кавички. Например, редиците: "Това е низ.", "1+23-34", "Hello\n" са низове.

*Забележка*: Операторът

```
cout << "Hello\n";
```

извежда върху екрана поздрава Hello и премества курсора на нов ред.

*Оператори*

В C++ има три групи оператори: аритметично-логически, управляващи и оператори за управление на динамичната памет.

- *аритметично-логически оператори*

Реализират основните аритметични и логически операции като: събиране (+), изваждане (-), умножение (*), деление (/), логическо И (&&, and), логическо ИЛИ (||, or) и др. В програмата Zad1.cpp използвахме аритметичните оператори * и +.

- *управляващи оператори*

Това са конструкции, които управляват изчислителния процес. Такива са условният оператор, операторът за цикъл, за безусловен преход и др.

- *операторите за управление на динамичната памет*

Те позволяват по време на изпълнение на програмата да бъде заделяна и съответно освобождавана динамична памет.

*Препинателни знаци*

Използват се ; < > { } ( ) и др. знаци.

**Разделяне на думите**

В C++ разделителите на думите са интервалът, вертикалната и хоризонталната табулации и знакът за нов ред.

**Коментари**

Коментарите са текстове, които не се обработват от компилатора, а служат само като пояснения за програмистите. В C++ има два начина за означаване на коментари. Единият начин е, текстът да се огради с /* и */. Тези коментари не могат да бъдат влагани. Другият начин са коментарите, които започват с // и завършват с края на текущия ред.

Коментарите са допустими навсякъде, където е допустим разделител.

*Забележка*: Не се препоръчва използването на коментари от вида // в редовете на директивите на компилатора.

## 2.3. Вход и изход

Програма Zad1.cpp намира периметъра и лицето само на правоъгълник със страни 2.3 и 3.7. Нека решим тази задача в общия случай.

**Задача 2.** Да се напише програма, която въвежда размерите на правоъгълник и намира периметъра и лицето му.

Програмата Zad2.cpp решава задачата.

```
// Program Zad2.cpp
#include <iostream.h>
int main()
{// въвеждане на едната страна
  cout << "a= ";
  double a;
  cin >> a;
 // въвеждане на другата страна
 cout << "b= ";
 double b;
 cin >> b;
 // намиране на периметъра
 double p;
 p = 2*(a+b);
 // намиране на лицето
 double s;
 s = a*b;
  // извеждане на периметъра
 cout << "p= " << p << "\n";
  // извеждане на лицето
 cout << "s= " << s << "\n";
 return 0;
}
```

Когато програмата бъде стартирана, върху екрана ще се появи подсещането

```
a=
```

което е покана за въвеждане размерите на едната страна на правоъгълника. Курсорът стои след знака =. Очаква се да бъде въведено число (цяло или реално), след което да бъде натиснат клавишът ENTER.

Следва покана за въвеждане на стойност за другата страна на правоъгълника, след което програмата ще изведе резултата и ще завърши изпълнението си.

Въвеждането на стойността на променливата a се осъществява с оператора за вход

cin >> a;

Обектът cin е името на стандартния входен поток, обикновено клавиатурата на компютъра. Изпълнението му води до пауза до въвеждане на число и натискане на клавиша ENTER. Нека за стойност на a е въведено 5.65, следвано от ENTER. В буфера на клавиатурата се записва

cin

| | | | | | |
|---|---|---|---|---|---|
| **5** | **.** | **6** | **5** | **\n** | |

След изпълнението на

cin >> a;

променливата a се свързва с 5.65, а в буфера на клавиатурата остава знакът \n, т.е.

cin

| | |
|---|---|
| **\n** | |

ОП

a

5.65

Въвеждането на стойността на променливата b се осъществява с оператора за вход

cin >> b;

Изпълнението му води до пауза до въвеждане на число и натискане на клавиша ENTER. Нека е въведено 8.3, следвано от ENTER. В буфера на клавиатурата имаме:

cin

| | | | | | |
|---|---|---|---|---|---|
| **\n** | **8** | **.** | **3** | **\n** | |

Изпълнението на оператора

cin >> b;

прескача знака \n, свързва 8.3 с променливата b, а в буфера на клавиатурата отново остава знакът \n, т.е.

cin

| | |
|---|---|
| **\n** | |

```
ОП
a         b
5.65      8.3
```

Чрез оператора за вход могат да се въвеждат стойности на повече от една променлива. Фиг. 2.6 съдържа по-пълно негово описание.

Входът от клавиатурата е буфериран. Това означава, че всяка редица от натиснати клавиши се разглежда като пакет, който се обработва чак след като се натисне клавишът ENTER.

**Оператор за вход >>**

*Синтаксис*

cin >> <променлива>;

където

- cin е обект (променлива) от клас (тип) istream, свързан с клавиатурата,
- <променлива> е идентификатор, дефиниран, като променлива от "допустим тип", преди оператора за въвеждане. (Типовете int, long, double са допустими).

*Семантика*

Извлича (въвежда) от cin (клавиатурата) поредната дума и я прехвърля в аргумента-приемник <променлива>. Конструкцията

cin >> <променлива>

е израз от тип istream със стойност левия му аргумент, т.е. резултатът от изпълнението на оператора >> е cin. Това позволява няколко думи да бъдат извличани чрез верига от оператори >>.

Следователно, допустим е следният по-общ синтаксис на >>:

cin >> <променлива> {>> <променлива>}$_{опц}$;

Операторът >> се изпълнява отляво надясно. Такива оператори се наричат лявоасоциативни. Така операторът

cin >> променлива$_1$ >> променлива$_2$ >> … >> променлива$_n$;

е еквивалентен на редицата от оператори:

cin >> променлива$_1$;
cin >> променлива$_2$;
…
cin >> променлива$_n$;

Освен това, ако операцията въвеждане е завършила успешно, състоянието на cin е true, в противен случай състоянието на cin е false.

*Пример:*

cin >> a >> b;

Фиг. 2.6 Оператор за вход >>

В случая

cin >> променлива$_1$ >> променлива$_2$ >> … >> променлива$_n$;

от Фиг. 2.6, настъпва пауза. Компилаторът очаква да бъдат въведени n *стойности* - за променлива$_1$, променлива$_2$,…, променлива$_n$, съответно и да бъде натиснат клавишът ENTER. Тези стойности трябва да бъдат въведени по подходящ начин (на един ред, на отделни редове или по няколко данни на последователни редове, слепени или разделени с интервали, табулации или знака за нов ред), като *стойност$_i$* трябва да бъде от тип, съвместим с типа на *променлива$_i$* (i = 1, 2, …, n).

*Пример:* Да разгледаме програмния фрагмент:

```
double a, b, c;
cin >> a >> b >> c;
```

Операторът за вход изисква да бъдат въведени три реални числа за a, b и c съответно. Ако се въведат

```
1.1    2.2     3.3 ENTER
```

променливата a ще се свърже с 1.1, b – с 2.2 и c – с 3.3. Същият резултат ще се получи, ако се въведе

```
1.1   2.2 ENTER
3.3 ENTER
```

или

```
1.1 ENTER
2.2   3.3 ENTER
```

или

```
1.1 ENTER
2.2 ENTER
3.7 ENTER
```

или даже ако се въведе

```
1.1   2.2   3.3   4.4 ENTER
```

В последния случай, стойността 4.4 ще остане необработена в буфера на клавиатурата и ще обслужи следващо четене, ако има такова. Този

начин на действие съвсем не е приемлив. Още по-лошо ще стане когато се въведат данни от неподходящ тип.

*Пример:* Да разгледаме фрагмента:

```
int a;
cin >> a;
```

Той дефинира целочислена променлива a (a е променлива от тип int), след което настъпва пауза в очакване да бъде въведено цяло число. Нека сме въвели 1.25, следвано от ENTER. Състоянието на буфера на клавиатурата е:

cin

| | | | | | |
|---|---|---|---|---|---|
| **1** | **.** | **2** | **5** | **\n** | |

Операторът

```
cin >> a;
```

свързва a с 1, но не прескача останалата информация от буфера и тя ще обслужи следващо четене, което води до непредсказуем резултат. Още по-неприятна е ситуацията, когато вместо цяло число за стойност на a се въведе някакъв низ, например one, следван от ENTER. В този случай, изпълнението на

```
cin >> a;
```

ще доведе до

cin

| | | | | |
|---|---|---|---|---|
| **o** | **n** | **e** | **\n** | **състояние** fail |

и

ОП

a

-

т.е. стойността на променливата a не се променя (остава неопределена), а буферът на клавиатурата изпада в състояние fail. За съжаление системата не извежда съобщение за грешка, което да уведоми за възникналия проблем.

Засега препоръчваме въвеждането на коректни входни данни. Преодоляването на недостатъците, илюстрирани по-горе, ще разгледаме в следващите части на книгата.

Вече използвахме оператора за изход. Фиг. 2.7 описва неговите синтаксис и семантика.

**Оператор за изход <<**

*Синтаксис*

cout << <израз>;

където

- cout е обект (променлива) от клас (тип) ostream, предварително е свързан с екрана на компютъра;
- <израз> е израз от допустим тип. Представата за израз продължава да бъде тази от математиката. Допустими типове са bool, int, short, long, double, float и др.

*Семантика*

Операторът << изпраща (извежда) към (върху) cout (екрана на компютъра) стойността на <израз>. Конструкцията

cout << <израз>

е израз от тип ostream и има за стойност първия му аргумент, т.е. резултатът от изпълнението на оператора << в горния случай е cout. Това позвалява чрез верига от оператори << да бъдат изведени стойностите на повече от един <израз>, т.е. допустим е следният по-общ синтаксис:

cout << <израз> $\{<< <израз>\}_{опц}$;

Операторът << се изпълнява отляво надясно (лявоасоциативен е). Така операторът

cout << $израз_1$ << $израз_2$ << … << $израз_n$;

е еквивалентен на редицата от оператори:

cout << $израз_1$;

cout << $израз_2$;

…

cout << $израз_n$;

*Пример:*

cout << “a= “ << a << “\n”;

Фиг. 2.7 Оператор за изход

## 2.4. Структура на програмата на C++

Когато програмата е малка, естествено е целият й код да бъде записан в един файл. Когато програмите са по-големи или когато се работи в колектив, ситуацията е по-различна. Налага се да се раздели кодът в отделни изходни (source) файлове. Причините, поради които се налага разделянето, са следните. Компилирането на файл отнема време и е глупаво да се чака компилаторът да превежда отново и отново код, който не е бил променян. Трябва да се компилират само файловете, които са били променяни.

Друга причина е работата в колектив. Би било трудно много програмисти да редактират едновременно един файл. Затова кодът на програмата се разделя така, че всеки програмист да отговаря за един или няколко файлове.

Ако програмата се състои от няколко файла, трябва да се каже на компилатора как да компилира и изгради цялата програма. Това ще направим в следващите раздели. Сега ще дадем най-обща представа за структурата на изходните файлове. Ще ги наричаме още модули.

Изходните файлове се организират по следния начин:

```
<изходен_файл> ::=   <заглавен_блок_с_коментари>
                     <заглавни_файлове>
                     <константи>
                     <класове>
                     <глобални_променливи>
                     <функции>
```

*Заглавен блок с коментари*

Всеки модул започва със заглавен блок с коментари, даващи информация за целта му, за използваните компилатор и операционна среда, за името на програмиста и датата на създаването на модула. Заглавният блок с коментари може да съдържа забележки, свързани с описания на структури от данни, аргументи, формат на файлове, правила, уговорки и друга информация.

*Заглавни файлове*

В тази част на модула са изброени всички необходими заглавни файлове. Например

```
#include <iostream.h>
```

```
#include <cmath.h>
```

Забелязваме, че за разделител е използван знакът за нов ред, а не ;.

*Константи*

В тази част се описват константите, необходими за модула. Вече имаме някаква минимална представа за тях. По-подробно описание на синтаксиса и семантиката им е дадена на Фиг. 2.8. За да бъде програмата по-лесна за четене и модифициране, е полезно да се дават символични имена не само на променливите, а и на константите. Това става чрез дефинирането на константи.

**Задача 3.** Да се напише програма, която въвежда радиуса на окръжност и намира и извежда дължината на окръжността и лицето на кръга с дадения радиус.

Една програма, която решава задачата е следната:

```
// Program Zad3.cpp
#include <iostream.h>
const double PI = 3.14159265;
int main()
{ double r;
  cout << “r= “;
  cin >> r;
  double p = 2 * PI * r;
  double s = PI * r * r;
  cout << “p=” << p << “\n”;
  cout << “s=” << s << “\n”;
  return 0;
}
```

В тази програма е дефинирана реална константа с име PI и стойност 3. 14159265, след което е използвано името PI.

**Дефиниране на константи**

*Синтаксис*

```
const  <име_на_тип> <име_на_константа> = <израз>;
```

където

const е запазена дума (съкращение от constant);

<име_на_тип> е идентификатор, означаващ име на тип;

<име_на_константа> е идентификатор, обикновено състоящ се от главни букви, за да се различава визуално от променливите.

<израз> е израз от тип, съвместим с <име_на_тип>.

*Семантика*

Свързва <име_на_константа> със стойността на <израз>. Правенето на опит да бъде променяна стойността на константата предизвиква грешка.

*Примери:*

```
const int MAXINT = 32767;
const double RI = 2.5 * MAXINT;
```

Фиг. 2.8 Дефиниране на константи

*Предимства на дефинирането на константите:*

- Програмите стават по-ясни и четливи.
- Лесно (само на едно място) се променят стойностите им (ако се налага).
- Вероятността за грешки, възможни при многократното изписване на стойността на константата, намалява.

*Забележка*: Тъй като в програмата Zad3.cpp е използвана само една функция (main), дефиницията на константата PI може да се постави във функцията main, преди първото нейно използване.

*Класове*

Тази част съдържа дефинициите на класовете, използвани в модула.

В езика C++ има стандартен набор от типове данни като int, double, float, char и др. Този набор може да бъде разширен чрез дефинирането на класове.

Дефинирането на клас въвежда нов тип, който може да бъде интегриран в езика. Класовете са в основата на обектно-ориетираното програмиране, за което е предназначен езикът C++.

Дефинирането и използването на класове ще бъде разгледано във втората част на курса.

*Глобални променливи*

Езикът поддържа глобални променливи. Те са променливи, които се дефинират извън функциите и които са “видими” за всички функции, дефинирани след тях. Дефинират се както се дефинират другите (локалните) променливи. Използването на много глобални променливи е лош стил за програмиране и не се препоръчва. Всяка глобална променлива трябва да е съпроводена с коментар, обясняващ предназначението й.

*Функции*

Всеки модул задължително съдържа функция main. Възможно е да съдържа и други функции. Тогава те се изброяват в тази част на модула. Ако функциите са подредени така, че всяка от тях е дефинирана преди да бъде извикана, тогава main трябва да бъде последна. В противен случай, в началото на тази част на модула, трябва да се **декларират** всички функции.

## Задачи

**Задача 1.** Кои от следните редици от знаци са идентификатори, кои не и защо?

| | | | | |
|---|---|---|---|---|
| а) a | б) x1 | в) $x_1$ | г) x’ | д) x1x2 |
| е) sin | ж) sin x | з) cos(x) | и) x-1 | к) 2a |
| л) min 1 | м) Beta | н) a1+a2 | о) k”m | п) sin’x |

**Задача 2.** Намерете синтактичните грешки в следващата програма:

```
include <iostream>
int Main()
{ cout >> “a, b =  “;
  cin << a, b;
  cout << “The product of “ << a << “and” << b << “is: “
       << a*b < “\n”
  return 0;
}
```

**Задача 3.** Напишете програма, която разменя стойностите на две числови променливи.

**Задача 4.** Напишете програма, която намира минималното (максималното) от две цели числа.

**Задача 5.** Напишете програма, която изписва с главни букви текста Hellow World.

Упътване: Голямата буква Н може да се представи така:

```
o   o
o   o
o   o
ooooo
o   o
o   o
o   o
```

и да се реализира по следния начин:

char* let_H = “o o\no o\no o\nooooo\no o\no o\no o\n”;

където char* означава тип низ.

## Допълнителна литература


1. Г. Симов, Програмиране на С++, София, СИМ, 1993.
2. К. Хорстман, Принципи на програмирането със С++, София, СОФТЕХ, 2000.
3. П. Лукас, Наръчник на програмиста, София, Техника, 1994.

# 3

# Скаларни типове данни

Езикът C++ е изключително мощен по отношение на типовете данни, които притежава. Най-общо, типовете му могат да бъдат разделени на: **вградени** и **абстрактни**.

Вградените типове са предварително дефинирани и се поддържат от неговото ядро.

Абстрактните типове се дефинират от програмиста. За целта се определят съответни класове.

Една непълна класификация на вградените типове данни е дадена на Фиг. 3.1.

Фиг. 3.1 Непълна класификация на вградени типове данни

Типовете булев, цял и символен се наричат **интегрални типове**, а типовете цял и реален – **числови типове**. Интегралните типове заедно с типа реален, се наричат **аритметични типове**, а типът изброен –

**потребителски дефиниран тип**. Скаларни са типовете данни, които се състоят от една компонента (число, знак и др.).

Съставни типове са онези типове данни, компонентите на които са редици от елементи.

Типът указател дава средства за динамично разпределение на паметта.

Типовете булев, цял и символен се наричат интегрални типове. Интегралните типове, заедно с типа реален, се наричат аритметични типове. Типът изброен се нарича потребителски дефиниран тип.

В тази глава ще разгледаме само някои скаларни типове данни.

Всеки тип се определя с **множество от допустими стойности** (множество от стойности) и **оператори и вградени функции**, които могат да се прилагат над елементите от множеството от стойностите му.

## 3.1. Логически тип

Нарича се още булев тип в чест на Дж. Бул, английски логик, поставил основите на математическата логика.

Типът е стандартен, вграден в реализацията. За означаването му се използва запазената дума bool (съкращение от boolean).

**Множество от стойности**

Състои се от два елемента – стойностите true (истина) и false (лъжа). Тези стойности се наричат още **булеви константи**.

<булева_константа> ::= true | false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа булев, се нарича **булева** или **логическа променлива** или **променлива от тип булев**. Дефинира се по обичайния начин.

*Примери:*

```
bool b1, b2;
bool b3 = false;
```

Дефиницията свързва булевите променливи с множеството от стойности на типа булев или с конкретна стойност от това множество като отделя по 1 байт оперативна памет за всяка от тях. Стойността на тази памет е неопределена или е булевата константа, свързана с дефинираната прменлива, в случай, че тя е инициализирана.

След дефиницията от примера по-горе, имаме:

```
ОП
b1        b2        b3
-         -         false
 1 байт   1 байт    1 байт
```

Стойността на клетките от паметта, именувани с b1 и b2, е неопределена, а тази на именуваната с b3 е false. В същност, вместо false в паметта е записан кодът (вътрешното представяне) на false - 0.

Вътрешните представяния на булевите константи са:

```
false    0
true     1
```

**Оператори и вградени функции**

**Логически оператори**

*Оператор за логическо умножение (конюнкция)*

Той е двуаргументен (бинарен) оператор. Означава се с and или && (за Visual C++ 6.0) и се дефинира по следния начин:

| A | B | A and B |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

Операторът се поставя между двата си аргумента. Такива оператори се наричат **инфиксни.**

*Оператор за логическо събиране (дизюнкция)*

Той също е бинарен, инфиксен оператор. Означава се с or или || (за Visual C++ 6.0) и се дефинира по следния начин:

| A | B | A or B |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

*Оператор за логическо отрицание*

Той е едноаргументен (унарен) оператор. Означава се с not или ! (за Visual C++ 6.0) и се дефинира по следния начин:

| A | not A |
|---|---|
| false | true |
| true | false |

Поставя се пред единствения си аргумент. Такива оператори се наричат **префиксни**.

*Забележка:* Може да няма разделител между оператора ! и константите true и false, т.е. записите !true и !false са допустими.

*Допълнение:* Смисълът на операторите and, or и not е разширен чрез разширяване смисъла на булевите константи. Прието е, че true е всяка стойност, различна от 0 и че false е стойността 0.

**Оператори за сравнение**

Над булевите данни могат да се извършват следните инфиксни оператори за сравнение:

| Оператор | Операция |
|---|---|
| == | сравнение за равно |
| != | сравнение за различно |
| > | сравнение за по-голямо |

| | |
|---|---|
| >= | сравнение за по-голямо или равно |
| < | сравнение за по-малко |
| <= | сравнение за по-малко или равно |

Сравняват се кодовете.

*Примери:*

false < true  е true

false > false  е false

true >= false  е true

*Въвеждане*

Не е възможно въвеждане на стойност на булева променлива чрез оператора >>, т.е. операторът

cin >> b1;

където b1 е булевата променлива, дефинирана по-горе, е недопустим.

*Извеждане*

Осъществява се чрез оператора

cout << <булева_константа>;

или по-общо

cout << <булев_израз>;

където синтактичната категория <булев_израз> е определена по-долу.

Извежда се кодът на булевата константа или кодът на булевата константа, която е стойност на <булев_израз>.

**Булеви изрази**

Булевите изрази са правила за получаване на булева стойност. Дефинират се *рекурсивно* по следния начин:

- Булевите константи са булеви изрази.
- Булевите променливи са булеви изрази.
- Прилагането на булевите оператори not (!), and (&&), or (||) над булеви изрази е булев израз.

- Прилагането на операторите за сравнение ==, !=, >, >=, <, <= към булеви изрази е булев израз.

*Примери:* Нека имаме дефиницията
bool b, b1, b2;
Следните изрази са булеви:

true b b1 b2 !false !!b !b1 || b2
!!!b && !!!!!b2 b < !b2 false >= b b1 == b2 > b b != b1

Тази дефиниция е непълна. Ще отбележим само, че сравнението на аритметични изрази чрез изброените по-горе оператори за сравнение, е булев израз. Освен това, аритметичен израз, поставен на място, където синтаксисът изисква булев израз, изпълнява ролята на булев израз. Това е резултат от разширяването смисъла на булевите константи true и false, чрез приемането всяка стойност, различна от 0 да се интерпретира като true и 0 да се интерпретира като false.

Засега отлагаме разглеждането на семантиката на булевите изрази. Това ще направим след разглеждане на аритметичните изрази.

## 3.2. Числени типове

### 3.2.1. Целочислени типове

Ще разгледаме целочисления тип int.

Типът е стандартен, вграден в реализацията на езика. За означаването му се използва запазената дума int (съкращение от integer).

#### Множество от стойности

Множеството от стойности на типа int зависи от хардуера и реализацията и не се дефинира от ANSI (American National Standarts Institute). Състои се от целите числа от някакъв интервал. За реализацията Visual C++ 6.0, това е интервалът [-2147483648, 2147483647].

Целите числа се записват като в математиката, т.е.

<цяло_число> ::= [+|-]<цяло_без_знак>

<цяло_без_знак> ::= <цифра>|
<цифра><цяло_без_знак>

<цифра> ::= 0|1| … |9.

Обикновено знакът + се пропуска.

*Допълнение:* Целите числа могат да са в десетична, осмична и шестнадесетична позиционна система. Осмичните числа започват с 0 (нула), а шестнадесетичните - с 0x (нула, x).

Елементите от множеството от стойности на типа int се наричат **константи от тип int**.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа int, се **нарича цяла променлива** или **променлива от тип int**. Дефинира се по обичайния начин. Дефиницията свързва променливата с множеството от стойности на типа int или с конкретна стойност от това множество като отделя по 4 байта (1 дума) оперативна памет за всяка от тях. Ако променливата не е била инициализирана, стойността на свързната с нея памет е неопределена, а в противен случай е указаната при инициализацията стойност.

*Примери:*

```
int i;
int j = 56;
```

След тези дефиниции, имаме:

```
ОП
i           j
-           56
4 байта   4 байта
```

**Оператори и вградени функции**

## Аритметични оператори

*Унарни оператори*

Записват се пред или след единствения си аргумент.

+, - са префиксни оператори. Потвърждават или променят знака на аргумента си.

*Примери:* Нека

```
int i = 12, j = -7;
```

Следните означения съдържат унарния оператор + или -:

```
-i    +j    -j    +i    -567
```

*Бинарни оператори*

Имат два аргумента. Следните аритметични оператори са инфиксни:

| Оператор | Операция |
|---|---|
| + | събиране |
| - | изваждане |
| * | умножение |
| / | целочислено деление |
| % | остатък от целочислено деление |

*Примери:*

```
15 - 1235 = -1220    13 / 5 = 2
15 + 1235 =  1250    13 % 5 = 3
-15 * 123 = -1845    23 % 3 = 2
```

*Забележка:* Допустимо е последователно използване на два знака за аритметични операции, но единият трябва да е унарен. Например, допустими са изразите 5-+4, 5+-4, имащи стойност 1, а също 5*-4, равно на -20.

## Логически оператори

Логическите оператори, реализиращи конюнкция, дизюнкция и отрицание, могат да се прилагат над целочислени константи. Дефинират

се по същия начин, както при булевите константи, но целите числа, които са различни от 0 се интерпретират true, а 0 – като false.

*Примери:*

123 and 0  е false
0 or 15  е true
not 67  е false

**Оператори за сравнение**

Над цели константи могат да се прилагат следните инфиксни оператори за сравнение:

| Оператор | Операция |
|---|---|
| == | сравнение за равно |
| != | сравнение за различно |
| > | сравнение за по-голямо |
| >= | сравнение за по-голямо или равно |
| < | сравнение за по-малко |
| <= | сравнение за по-малко или равно |

Наредбата на целите числа е като в математиката.

*Примери:*

123 < 234  е true
-123456 > 324  е false
23451 >= 0  е true

**Вградени функции**

В езика C++ има набор от вградени функции. Обръщението към такива функции има следния синтаксис:

<име_на_функция>(<израз>, <израз>, …, <израз>)
и връща стойност от типа на функцията.

Тук ще разгледаме само едноаргументната целочислена функция abs.

abs(x) – намира |x|, където x е цял израз

(в частност цяла константа).

*Примери:*

abs(-1587) = 1587  abs(0) = 0  abs(23) = 23

За използването на тази функция е необходимо в частта на заглавните файлове да се включи заглавният файл math.h чрез:

```
#include <math.h>
```

Библиотеката math.h съдържа богат набор от функции. В някой реализации тя има име math или cmath.

**Въвеждане**

Реализира се по стандартния и разгледан вече начин.

*Пример:* Ако

```
int i, j;
```

операторът

```
cin >> i >> j;
```

въвежда стойности на целите променливи i и j. Очаква се въвеждане от стандартния входен поток на две цели константи от тип int, разделени с интервал, знаците за хоризонтална или вертикална табулация или знака за преминаване на нов ред.

**Извеждане**

Реализира се чрез оператора

```
cout << <цяла_константа>;
```

или по-общо

```
cout << <цял_израз>;
```

В текущата позиция на курсора се извежда константата или стойността на целия израз. Използва се минималното количество позиции, необходими за записване на цялото число.

*Пример:* Нека имаме дефиницията

```
int i = 1234, j = 9876;
```

Операторът

```
cout << i << j << "\n";
```

извежда върху екрана стойностите на i и j, но слепени

12349876

Този изход не е ясен. Налага се да се извърши **форматиране** на изхода. То се осъществява чрез подходящи **манипулатори**.

**Манипулатор setw**

Setw е вградена функция.

*Синтаксис*

setw(<цял_израз>)

*Семантика*

Стойността на <цял_израз> задава широчината на полето на **следващия** изход.

*Пример:* Операторът

cout << setw(10);

не извежда нищо. Той "манипулира" следващото извеждане като указва, че в поле с широчина 10 отдясно приравнена, ще бъде записана следващата извеждана цяла константа.

**Забележка:** Този манипулатор важи само за първото след него извеждане.

*Пример:* Нека

ОП

i j

1234 9876

Операторът

cout << setw(10) << i << j << "\n";

извежда отново стойностите на i и j слепени, като 1234 се предшества от 6 интервала, т.е.

******12349876

където интервалът е означен със знака *.

Операторът

cout << setw(10) << i << setw(10) << j << "\n";

извежда

******1234******9876

**Манипулатори dec, oct и hex**

Целите числа се извеждат в десетична позиционна система. Ако се налага изходът им да е в осмична или шестнадесетична позиционна система, се използват манипулаторите oct и hex съответно. Всеки от тях е в сила, докато друг манипулатор за позиционна система не е указан за следващ извод. Връщането към десетична позиционна система се осъществява чрез манипулатора dec.

dec – манипулатор, задаващ всички следващи изходи на цели числа (докато не е указан друг манипулатор, променящ позиционната система) да са в десетична позиционна система;

oct – манипулатор, задаващ всички следващи изходи на цели числа (докато не е указан друг манипулатор, променящ позиционната система) да са в осмична позиционна система;

hex - манипулатор, задаващ всички следващи изходи на цели числа (докато не е указан друг манипулатор, променящ позиционната система) да са в шестнадесетична позиционна система.

*Пример:* Нека имаме

ОП

i j

12 23

След изпълнението на операторите

```
cout << setw(10) << dec << i << setw(10) << j << “\n”;
cout << setw(10) << oct << i << setw(10) << j << “\n”;
cout << setw(10) << hex << i << setw(10) << j << “\n”;
```

имаме:

```
********12********23
********14********27
*********c********17
```

*Забележка:* Преди използване на манипулаторите е необходимо да се включи заглавният файл iomanip.h, т.е. в частта за заглавни файлове да се запише директивата #include <iomanip.h>

**Други целочислени типове**

Други цели типове се получават от int като се използват модификаторите short, long, signed и unsigned. Тези модификатори доопределят някои аспекти на типа int.

За реализацията Visual C++ 6.0 са в сила:

| Тип | Диапазон | Необходима памет |
|---|---|---|
| short int | -32768 до 32767 | 2 байта |
| unsigned short int | 0 до 65535 | 2 байта |
| long int | -2147483648 до 2147483647 | 4 байта |
| unsigned long int | 0 до 4294967295 | 4 байта |
| unsigned int | 0 до 4294967295 | 4 байта |

Запазената дума int при тези типове се подразбира и може да бъде пропусната. Типовете short int (или само short) и long int (или само long) са съкратен запис на signed short int и signed long int.

### 3.2.2. Реални типове

Ще разгледаме реалния тип double.

Типът е стандартен, вграден във всички реализации на езика.

**Множество от стойности**

Множеството от стойности на типа double се състои от реалните числа от $-1.74*10^{308}$ до $1.7*10^{308}$ . Записват се във два формата – като числа с фиксирана и като числа с плаваща запетая (експоненциален формат).

```
<реално_число> ::= <цяло_число>.<цяло_число_без_знак>|
                   <цяло_число>E<порядък>|
                   <цяло_число>.<цяло_число_без_знак>E<порядък>|
<порядък> ::= <цяло_число>
```

При експоненциалния формат може да се използва и малката латинска буква е.

*Примери:* Следните числа

123.3454  -10343.034  123E13  -1.23e-4

са коректно записани реални числа.

Смисълът на експоненциалния формат е реално число, получено след умножаване на числото пред E (e) с 10 на степен числото след E (e).

*Примери:* 12.5E4 е реалното число 125000.0, а –1234.025e-3 е реалното число –1.234025.

Елементите от множеството от стойности на типа double се наричат **реални константи** или по-точно **константи от реалния тип double**.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа double, се нарича **реална променлива** или **променлива от тип double**.

Дефинира се по обичайния начин. Дефиницията свързва реалните променливи с множеството от стойности на типа double или с конкретна стойност от това множество, като отделя по 8 байта оперативна памет за всяка от тях. Ако променливата не е била инициализирана, стойността на свързаната с нея памет е неопределена, а в противен случай е указаната при инициализацията стойност.

*Примери:*

```
double i;
double j = 5699.876;
```

След тази дефиниция, имаме:

```
ОП
   i              j
   -           5699.876
8 байта        8 байта
```

Оператори и вградени функции

**Аритметични оператори**

*Унарни оператори*

+, -  Префиксни са. Потвърждават или променят знака на аргумента си.

*Примери:* Нека

```
double i = 1.2, j = -7.5;
```

Следните конструкции съдържат унарен оператор + или -:

-i  +j  -j  +i  -56.7

*Бинарни оператори*

Имат два аргумента. Следните аритметичните оператори са инфиксни:

| Оператор | Операция |
|---|---|
| + | събиране |
| - | изваждане |
| * | умножение |
| / | деление (поне единият аргумент е реален) |

*Примери:*

15.3 – 12.2 = 3.1  13.0 / 5 = 2.6

15 + 12.35 =  27.35  13 / 5.0 = 2.6

-1.5 * 12.3 = -18.45

**Логически оператори**

Логическите оператори, реализиращи операциите конюнкция, дизюнкция и отрицание, могат да се прилагат над реални константи. Дефинират се по същия начин, както при булевите константи, като реалните числа, които са различни от 0.0 се интерпретират като true, а 0.0 – като false.

*Примери:*

123.6 and 0.0  е false

0.0 or 15.67  е true

not 67.7  е false

**Оператори за сравнение**

Над реални данни могат да се прилагат следните инфиксни оператори за сравнение:

| Оператор | Операция |
|---|---|

| | |
|---|---|
| == | сравнение за равно |
| != | сравнение за различно |
| > | сравнение за по-голямо |
| >= | сравнение за по-голямо или равно |
| < | сравнение за по-малко |
| <= | сравнение за по-малко или равно |

Наредбата на реалните числа е като в математиката.

*Примери:*

```
123.56 < 234.09            е true
-123456.9888 > 324.0098    е false
23451.6 >= 0               е true
```

*Допълнение:* Сравнението за равно на две реални числа x и y се реализира обикновено чрез релацията: $|x - y| < \varepsilon$, където $\varepsilon = 10^{-14}$ за тип double. По-добър начин е да се използва релацията:

$$\frac{|x - y|}{\max\{|x|, |y|\}} \leq \varepsilon$$

**Вградени функции**

При цял или реален аргумент, следните функции връщат реален резултат от тип double:

| Функция | Намира |
|---|---|
| sin(x) | Синус, sin x, x е в радиани |
| Cos(x) | косинус, cos x, x е в радиани |
| tan(x) | тангенс, tg x, x е в радиани |
| asin(x) | аркуссинус, arcsin x $\in$ [-π/2, π/2], x $\in$ [-1, 1] |
| acos(x) | аркускосинус, arccos x $\in$ [0, π], x $\in$ [-1, 1] |
| atan(x) | аркустангенс, arctg x $\in$ (-π/2, π/2) |
| exp(x) | експонента, $e^x$ |
| log(x) | натурален логаритъм, ln x, x > 0 |
| log10(x) | десетичен логаритъм, lg x, x > 0 |

| | |
|---|---|
| sinh(x) | хиперболичен синус, sh x |
| cosh(x) | хиперболичен косинус, ch x |
| tanh(x) | хиперболичен тангенс, th x |
| ceil(x) | най-малкото цяло ≥ x, преобразувано в тип double |
| Floor(x) | най-голямото цяло ≤ x, преобразувано в тип double |
| fabs(x) | абсолютна стойност на x, \|x\| |
| sqrt(x) | Корен квадратен от x, x ≥ 0 |
| pow(x, n) | степенуване, $x^n$ (x и n са реални от тип double). |

*Примери:* ceil(12.345) = 13.0 ceil(-12.345) = -12.0
ceil(1234) = 1234.0 ceil(-1234) = -1234.0
floor(12.345) = 12.0 floor(-12.345) = -13.0
floor(123) = 123.0 floor(-123) = -123.0
fabs(123) = 123.0 fabs(-1234) = 1234.0
sin(PI/6) намира sin(30°), където PI = 3.142857

Тези функции се намират в библиотеката math.h и за да бъдат използвани е необходимо в частта на заглавните файлове да бъде включена директивата:

```
#include <math.h>
```

**Задача 4.** Да се напише програма, която намира функцията скобка от дадено реално число.

Следната програма решава задачата:

```
#include <iostream.h>
#include <math.h>
int main()
{cout << “x= “;
 double x;
 cin >> x;
 int y;
 y = floor(x);
 cout << y << “\n”;
 return 0;
}
```

Компилаторът издава предупреждение, че на линия 8 се извършва **преобразуване** на типа double в тип int, което може да доведе до загуба на информация. Но това е един начин за преобразуване на тип (чрез оператор за присвояване).

В езика C++ са реализирани и други начини за преобразуване на типове:

а) (тип)<израз>

където тип е име на тип.

Пресмята се стойността на <израз> и получената константа се преобразува в указания в скобите тип.

*Примери:*

(int)(1.52 + 56.2) = 57

(double)(123+18) = 141.0

б) static_cast< тип >(<израз>)

където тип е име на тип.

Пресмята се стойността на <израз> и получената константа се преобразува в указания в скобите < … > тип.

*Примери:*

Static_cast<int>(1.52 + 56.2) = 57

Static_cast<double>(123+18) = 141.0

*Забележка:* При явното преобразуване на типове, не се получава предупреждение от компилатора за загуба на точност.

**Въвеждане**

Реализира се по стандартния начин.

*Пример:* Ако

double x, y;

операторът

cin >> x >> y;

въвежда стойности на x и y.

За разделител се използват: интервалът, знаците за вертикална и хоризонтална табулация и знакът за преминаване на нов ред. Ако за

някоя реална променлива е въведена цяла константа, извършва се конвертиране в реален тип, след което полученото реално число се записва в отделената за променливата памет.

**Извеждане**

Реализира се чрез оператора

cout << <реална_константа>;

или по-общо

cout << <реален_израз>;

В текущата позиция на курсора се извежда реалната константа или стойността на реалния израз. Използва се минималното количество позиции, необходими за записване на реалното число.

*Пример:* Нека имаме дефиницията

double x = 12.34, y = 9.876;

Операторът

cout << x << y << “\n”;

извежда върху екрана стойностите на x и y слепени

12.349.876

Този изход не е ясен. Налага се да се извърши форматиране на изхода. То се осъществява чрез подходящи манипулатори.

**Манипулатор setw**

Setw е функция.

*Синтаксис*

setw(<цял_израз>)

*Семантика*

Задава широчината на **следващото** изходно поле.

*Пример:* Операторът

cout << setw(12);

не извежда нищо. Той “манипулира” следващото извеждане като указва, че в поле с широчина 12 отдясно приравнена, ще бъде записана следващата извеждана реална константа.

Този манипулатор важи само за първото след него извеждане.

*Пример:* Нека

ОП

x y
1.56 -2.36

Операторът

cout << setw(10) << x << y << “\n”;

извежда отново стойностите на x и y слепени, като 1.56 се предшества от 6 интервала, т.е.

******1.56-2.36

където интервалът е означен със знака *.

Операторът

cout << setw(10) << x << setw(10) << y << “\n”;

извежда

******1.56*****-2.36

При реалните данни се налага използването и на манипулатор за задаване на броя на цифрите след десетичната точка.

**Манипулатор setprecision**

*Синтаксис*

setprecision(<цял_израз>)

*Семантика*

Стойността на аргумента на тази функция задава броя на цифрите, използвани при извеждане на следващите реални числа, а в съчетание със “загадъчното” обръщение setiosflags(ios::fixed), задава броя на цифрите след десетичната точка на извежданите реални числа.

*Пример 1:* Операторът

cout << setprecision(3);

не извежда нищо. Той указва, че следващите реални константи ще се извеждат с 3 значещи цифри.

Нека в ОП имаме:

ОП

x

21.5632

Операторът

cout << setprecision(3) << setw(10) << x << “\n”;

извежда

******21.6

като извършва закръгляване.

А оператора

```
cout << setprecision(2) << setw(10) << x << "\n";
```

извежда

********22

*Пример 2:* Операторът

```
cout << setiosflags(ios::fixed) << setprecision(3);
```

не извежда нищо. Той указва, че следващите реални константи ще се извеждат с точно 3 цифри след десетичната точка.

Нека в ОП имаме:

ОП

x

21.5632

Операторът

```
cout << setiosflags(ios::fixed)
     << setprecision(3)
     << setw(10) << x << "\n";
```

извежда

****21.563

А операторът

```
cout << setiosflags(ios::fixed)
     << setprecision(1)
     << setw(10) << x << "\n";
```

извежда

******21.6

като извършва закръгляване.

*Забележка:* За щастие, манипулаторите setprecision(…) и setiosflags (…) са в сила не само за първата извеждана стойност, но и за всички следващи, докато с нов setprecition(…) не се промени точността и с обръщение към resetiosflags(ios::fixed) не се отмени валидността на setiosflags(ios::fixed).

*Пример:* Изпълнението на програмата

```
#include <iostream.h>
#include <iomanip.h>
int main()
{double a = 209.5, b = 63.75658;
```

```
    cout  << setprecision(3)
          << setiosflags(ios::fixed);
    cout << setw(10) << a << "\n";
    cout << setw(10) << b << "\n";
    cout << resetiosflags(ios::fixed);
    cout << setw(20) << a << "\n";
    cout << setw(20) << b << "\n";
    return 0;
    }
```

извежда

```
***209.500
****63.757
*****************210
****************63.8
```

*Допълнение:* Точността на типа double е около 15 значещи цифри. За да се убедите в това, изпълнете следната програма:

```
#include <iostream.h>
int main()
{ double a = 5e14;
  double b = a - 0.1;
  double c = a - b;
  cout << c << "\n";
  return 0;
}
```

Стойността на променливата c трябва да бъде 0.1, а програмата намира за такава 0.125. Причината е в броя на значещите цифри на b.

**Други реални типове**

В езика C++ има и друг реален тип, наречен float. Различава се от типа double по множеството от стойностите си и заеманата памет.

Множеството от стойности на типа float се състои от реалните числа от диапазона от $-3.4*10^{38}$ до $3.4*10^{38}$. За записване на константите от този диапазон са необходими 4 байта ОП.

Броят на значещите цифри при този тип е около 7. За да се убедите в това, изпълнете следната програма:

```
#include <iostream.h>
int main()
{ float a = 5e6;
  float b = a - 0.1;
  float c = a - b;
  cout << c << "\n";
  return 0;
}
```

Стойността на c трябва да е 0.1, а след изпълнение на горната програма се получава 0. Компилаторът предупреждава, че на линия 4 става преобразуване от тип float в тип double.

Реална константа от диапазона на тип float, но с повече от 7 значещи цифри се приема от компилатора за реално число от тип double и присвояването му на променлива от тип float издава предупреждението, че се извършва преобразуване от тип double в тип float, което може да доведе до загуба на точност.

Точността е причината заради, която препоръчваме използването на типа double.

### 3.2.3. Аритметични изрази

Аритметичните изрази са правила за получаване на числови константи. Има два вида аритметични изрази: цели и реални.

<аритметичен_израз> ::= <цял_израз>|<реален_израз>

**Цели аритметични изрази**

Целите аритметични изрази са правила за получаване на константи от тип int или разновидностите му. Дефинират се рекурсивно по следния начин:

- Целите константи са цели аритметични изрази.

*Примери:* 123 -2345 -32767
0 22233345 -87

са цели аритметични изрази.

- Целите променливи са цели аритметични изрази.

*Примери:* Ако имаме дефиницията:

```
int i, j;
short p, q, r;
```

```
i     j
```

са цели аритметични изрази от тип int, а

```
p     q     r
```

са цели аритметични изрази от тип short.

- Прилагането на унарните оператори + и – към цели аритметични изрази е цял аритметичен израз.

*Примери:* -i  +j  -j

са цели аритметични изрази от тип int, а

```
+p    -p    +r    -q
```

са цели аритметични изрази от тип short.

- Прилагането на бинарните аритметични оператори +, -, *, / и % към цели аритметични изрази, е цял аритметичен израз.

*Примери:* i % 10 + j * i - p  -i + j / 5

са цели аритметични изрази от тип int,

r – p / 12 – q  r % q – p  r + p – q

са цели аритметични изрази от тип short.

- Цялата функция abs, приложена над цял аритметичен израз, е цял аритметичен израз от тип int.

*Примери:* abs(i+j) и abs(p-r) са цели аритметични изрази от тип int.

**Реални аритметични изрази**

Реалните аритметични изрази са правила за получаване на константа от тип double или float. Дефинират се рекурсивно по следния начин:

- Реалните константи са реални аритметични изрази.

*Примери:* 1.23e-3  –2345e2  –3.2767  0.0

222.33345  -8.7009

са реални аритметични изрази.

*Забележка:* Реална константа от диапазона на тип float, но с повече от 7 значещи цифри се приема от компилатора за реално число от тип double.

- Реалните променливи са реални аритметични изрази.

*Примери:* Ако имаме дефинициите:

```
double i, j;
float p, q, r;
```

```
i     j
```

са реални аритметични изрази от тип double, а

```
p     q     r
```

са реални аритметични изрази от тип float.

- Прилагането на унарните оператори + или – към реален аритметичен израз е реален аритметичен израз.

*Примери:* -i  +j  -j

са реални аритметични изрази от тип double, а

+p  -p  +r  -q

са реални аритметични изрази от тип float.

- Прилагането на бинарните аритметични оператори +, -, * и / към аритметични изрази, поне един от които е реален, е реален аритметичен израз.

*Пример:* i/10 + j*i  - p  -i + j/5

са реални аритметични изрази от тип double, а

–p/12 – q  r/q – p  r + p – q

са реални аритметични изрази от тип float.

- Реалните функции: sin, cos, tan, asin, acos, atan, exp, log, log10, sinh, cosh, tanh, ceil, floor, fabs, sqrt и pow, приложени над реални или цели аритметични изрази, са реални аритметични изрази от тип double.

*Примери:* fabs(i+j)  sin(i-p)  cos(p/r-q)  floor(p)
ceil(r-p+i)  exp(p)  log(r-p*q)

### Семантика на аритметичните изрази

Аритметичният израз е правило за получаване на цяла или реална константа. За пресмятане на стойността му се използва следният приоритет на операторите и вградените функции:

1. Вградени функции
2. Действията в скобите
3. Оператори в следния приоритет

- +, - (унарни) най-висок
- *, /, %
- +, - (бинарни)
- <<, >> най-нисък

*Забележка 1:* Операторите >> и << са за побитови измествания надясно и наляво съответно. Те са предефинирани за реализиране на входно/изходни операции. В случая имаме предвид тази тяхна употреба.

*Забележка 2:* Инфиксните оператори, които са разположени на един и същ ред, са с еднакъв приоритет. Тези оператори се изпълняват отляво надясно (ляво асоциативни). Унарните оператори се изпълняват отдясно наляво (дясно асоциативни).

*Пример:* Нека имаме дефиницията

```
double x = 23.56, y = -123.5;
```

Изпълнението на оператора

```
cout << sin(x) + ceil(y)*x – cos(y);
```

ще се извърши по следния начин: отначало ще се пресметнат стойностите на sin(x), ceil(y) и cos(y), след това ще изпълни операторът * над стойността на ceil(y) и x, полученото ще събере със стойността на sin(x), след което от полученото реално число ще се извади пресметнатата вече стойност на cos(y). Накрая върху екрана ще се изведе получената реална константа.

Аритметичните изрази могат да съдържат операнди от различни типове. За да се пресметне стойността на такъв израз, автоматично се извършва преобразуване на типовете на операндите му. Без загуба на точността се осъществяват следните преобразувания:

| Тип | Преобразува се до |
|---|---|
| bool | всички числови типове |
| short | int |
| unsigned short | unsigned int |
| float | double |

т.е.конвертира се от “по-малък” тип (в байтове) към “по-голям” тип.

**Семантика на булевите изрази**

Булевите изрази са правила за получаване на булева стойност. За пресмятане на стойностите им се използва следният приоритет на операторите и вградените функции:

1. Вградени функции
2. Действията в скобите
3. Оператори в следния приоритет

- !, not, +, - (унарни) най-висок
- *, /, %
- +, - (бинарни)
- >> << (вход/изход)
- <, <=, >, >=
- ==, !=
- &&
- || най-нисък

Освен това, при булеви изрази от вида A && B, изразът B се пресмята само ако изразът A има стойност true, при булеви изрази от вида A || B, изразът B се пресмята само ако A има стойност false. Този начин на пресмятане на някои булеви изрази се нарича **отложено пресмятане**.

*Забележка:* Инфиксните оператори, които са разположени на един и същ ред са с еднакъв приоритет. Тези оператори се изпълняват отляво надясно, т.е. ляво асоциативни са. Унарните оператори се изпълняват отдясно наляво, т.е. дясно асоциативни са.

*Примери:* Нека имаме дефинициите:

```
double x = 23.56, y = -123.5;
bool b1, b2, b3;
b1 = true;
b2 = !b1;
b3 = b1||b2;
```

а) Изпълнението на оператора

```
cout << sin(x) + ceil(y) * x > 12398;
```

ще сигнализира грешка – некоректни аргументи на <<. Това е така, заради нарушения приоритет. Операторът << е с по-висок приоритет от този на операторите за сравнение. Налага се вторият аргумент на оператора << да бъде ограден в скоби. Операторът

```
cout << (sin(x) + ceil(y) * x > 12398);
```

също вече работи добре.

б) Изпълнението на оператора

```
cout << b1 && b2 || b3 << "\n";
```

също съобщава грешка – некоректни аргументи на <<. Отново е нарушен приоритетът на операциите. Налага се аргументът b1 && b2 || b3 на << да се огради в скоби, т.е.

```
cout << (b1 && b2 || b3) << "\n";
```

вече работи добре.

## Задачи върху типовете булев, цял и реален

**Задача 5.** Кои от следните редици от знаци са числа в C++?

| | | | |
|---|---|---|---|
| а) 061 | б) –31 | в) 1/5 | г) +910.009 |
| д) VII | е) 0.(3) | ж) sin(0) | з) 134+12 |

Редиците от знаци а), б), г) са числа.

**Задача 6.** Да се запишат на C++ следните числа:

| | | | |
|---|---|---|---|
| а) 6! | б) LXXIV | в) –0,4(6) | г) 138,2(38) |
| д) 11/4 | е) $\pi$ | ж) $1,2\ .10^{-1}$ | з) $-23,(1)\ .10^{2}$ |

В дробната част да се укажат до 4 цифри.

| | | | |
|---|---|---|---|
| а) 120 | б) 74 | в) –0.4667 | г) 138.2384 |
| д) 2.7500 | е) 3.1429 | ж) 0.1200 | з) –2311.1111 |

**Задача 7.** Да се запишат на C++ следните математически формули:

а) $a + b - a^2.b^3.c^4$ б) $\frac{a.b}{c} + \frac{c}{a.b}$

в) $(1 + \frac{x}{1!} + \frac{x^2}{2!}).(1 + \frac{x^3}{3!} + \frac{x^5}{5!})$ г) $\sqrt{1 + \sqrt{2 + \sqrt{3 + \sqrt{4}}}}$

а) a + b – a * a * b * b * b * c * c * c * c

б) (a * b) / c + c / (a * b)

в) (1 + x + x*x/2)*(1 + x*x*x/6 + x*x*x*x*x/120)

или

(1 + x + pow(x,2))/(1 + pow(x, 3)/6 + pow(x, 5)/120)

г) sqrt(1 + sqrt(2 + sqrt(3 + sqrt(4))))

**Задача 8.** Какво ще бъде изведено след изпълнението на следната програма:

```
#include <iostream.h>
#include <math.h>
int main()
{ cout << "x=";
  double x;
  cin >> x;
  bool b;
  b = x < ceil(x);
  cout << "x=";
  cin >> x;
  b = b && (x < floor(x));
  cout << "b= " << b << "\n";
  return 0;
 }
```

ако като вход бъдат зададени числата

а) 2.7 и 0.8 б) 2.7 и –0.8 в) –2.7 и –0.8?

а) Тъй като булевият израз 2.7 < ceil(2.7) има стойност true, а true && (0.8 < floor(0.8)) - е false, ще бъде изведено 0 (false).

**Задача 9.** Какъв ще е резултатът от изпълнението на програмата

```
#include <iostream.h>
```

```
int main()
{int a, b;
 cin >> a >> b >> a >> b >> a;
 cout << a << " " << b << " "
      << a << " " << b << " "
      << a << "\n";
 return 0;
}
```

ако като вход са зададени числата 1, 2, 3, 4 и 5?

След обработката на дефиницията int a, b; за променливите a и b са отделени по 4 байта ОП, т.е.

ОП

a b

- -

а след изпълнението на оператора за четене >>, a и b получават отначало стойностите 1 и 2. След това стойностите им се променят на 3 и 4 съответно. Най-накрая a става 5, т,е,

a b

5 4

Тогава програмата извежда

5 4 5 4 5

**Задача 10.** Да се запише булев израз, който има стойност true, ако посоченото условие е в сила, и стойност false, ако условието не е в сила.

а) цялото число a се дели на 5;

б) точката x принадлежи на отсечката [2, 6];

в) точката x не принадлежи на отсечката [2, 6];

г) точката x принадлежи на отсечката [2, 6] или на отсечката [-4, -2];

д) поне едно от числата a, b и c е отрицателно;

е) числата a, b и c са равни помежду си.

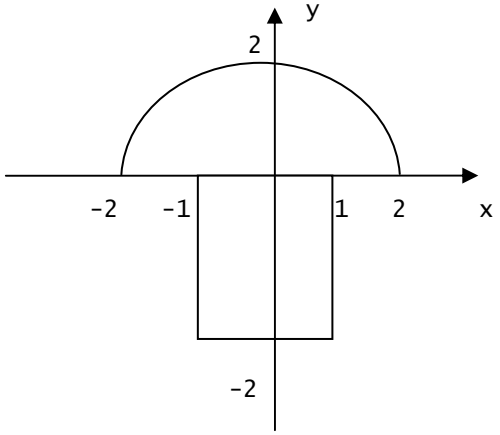
а) a % 5 == 0

б) x >= 2 && x <= 6

в) x < 2 || x > 6 или !( x >= 2 && x <= 6)

г) x >= 2 && x <= 6 || x >= -4 && x <= -2
д) a < 0 || b < 0 || c < 0
е) a == b && a == c

**Задача 11.** Да се напише програма, която въвежда координатите на точка от равнината и извежда 1, ако точката принадлежи на фигурата по-долу и – 0, в противен случай.




```
#include <iostream.h>
#include <math.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 cout << "y= ";
 double y;
 cin >> y;
 bool b1 = x*x + y*y <= 4 && y >= 0;
 bool b2 = fabs(x) <= 1 && y < 0 && y >= -2;
 cout << (b1 || b2) << "\n";
 return 0;
}
```

## Задачи

**Задача 1.** Да се запишат на езика C++ следните математически формули:

а) $\ln(x^4 + e^x + 10)$ б) $\log_5(x^4 + x^2 + 8)$

в) $(\text{acos } x - [x] - 1)^3 * e^{x+1,23}$ г) $\dfrac{a - b.c}{(\text{arctg } a - \text{sh } b + \text{ch } c)^4}$

**Задача 2.** Кои от следните редици от символи са правилно записани изрази на езика C++:

а) 1 + |y| г) 1 + sqrt(sin((u+v)/10))

б) -abs(x) + sin z д) -6 + xy

в) abs(x) + cos(abs(y - 1.7)) е) 1/-2 + Beta.

**Задача 3.** Да се запишат в традиционна (математическа) форма следните изрази, записани на езика C++:

а) sqrt(a+b) - sqrt(a-b) в) x*y/(u+v)-(u-v)/y*(a+b)

б) a + b/(c+d)-(a+b)/c+d г) 1+exp(cos((x+y)/2)).

**Задача 4.** Да се пресметне стойността на израза:

а) cos(0) + abs(1/(1/3-1))

б) abs(a-10) + sin(a-1), за a = 1;

в) cos(-2+2*x) +sqrt(fabs(x-5)), за x = 1;

г) sin(sin(x*x-1)*sin(x*x-1)) + cos(x*x*x-1)*abs(x-2), за x = 1;

д) sin(sin(x*x-1))+cos(x*x*x-1)*cos(abs(x-2)-1)/y*a + sqrt(abs(y)-x), за x = 1, y = -2, a = 2.

**Задача 5.** В аритметичния израз

а) a/b*c/d*e/f*h

б) a+b/x-2*y

в) a+b/x-2*y

да се поставят скоби така, че полученият израз да съответствува на математическата формула:

а) $\dfrac{a}{b \,.\, \dfrac{c}{d \,.\, \dfrac{e}{f \,.\, h}}}$ б) $\dfrac{a + b}{x - 2.y}$ в) $a + \dfrac{b}{x - 2} \,.\, y$

**Задача 6.** Да се напише израз на езика C++, който да изразява:

а) периметъра на квадрат с лице, равно на a;

б) лицето на равностранен триъгълник с периметър, равен на p.

**Задача 7.** Да се напише програма, която пресмята стойността на v1, където

а) $v1 = m + \cfrac{n}{p + \cfrac{q}{r + \cfrac{s}{t}}}$ б) $v1 = \dfrac{\sin(\sin(\sin x)) + \cos(\cos(\cos x))}{|\ln x| + |\cos x| + e^x}$

**Задача 8.** Да се пресметне стойността на израза:

а) pow(x, 2) + pow(y, 2) <= 4 при x = 0.6, y = -1.2

б) p%7 == p/5 - 2 при p = 15

в) floor(10*k+16.3)/2 == 0 при k = 0.185

г) !((k+325)%2 == 1) при k = 28

д) u*v != 0 && v > u при u = 2, v = 1

е) x || !y при x = false, y = true.

**Задача 9.** Да се запише булев израз, който да има стойност истина, ако посоченото условие е вярно и стойност - лъжа, ако условието не е вярно:

а) цялото число p се дели на 4 или на 7;

б) уравнението $a.x^2 + b.x + c = 0$ ($a \neq 0$) няма реални корени;

в) точка с координати (a,b) лежи във вътрешността на кръг с радиус 5 и център (0,1).

г) точка с координати (a,b) лежи извън кръга с център (c,d) и радиус f;

д) точка принадлежи на частта от кръга с център (0,0) и радиус 5 в трети квадрант;

е) точка принадлежи на венеца с център (0,0) и радиуси 5 и 10;

ж) x принадлежи на отсечката [0, 1];

з) x = max{a, b, c}

и) x != max{a, b, c} (операторът ! да не се използва);

к) поне една от булевите променливи x и y има стойност true;

л) и двете булеви променливи x и y имат стойност true;

м) нито едно от числата a, b и c е положително;

н) цифрата 7 влиза в записа на положителното трицифрено число p;

о) цифрите на трицифреното число m са различни;

п) поне две от цифрите на трицифреното число m са равни помежду си.

## Допълнителна литература


1. К. Хорстман, Принципи на програмирането със C++, София, СОФТЕХ, 2000.
2. П. Лукас, Наръчник на програмиста, София, Техника, 1994.
3. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.

# 4

# Основни структури за управление на изчислителния процес

В тази глава ще разгледаме управляващите оператори в езика. Ще ги наричаме само оператори.

## 4.1. Оператор за присвояване

Това е един от най-важните оператори на езика. Вече многократно го използвахме, а също в глава 2 описахме неговите синтаксис и семантика. В тази глава ще го разгледаме по-подробно. Ще напомним неговите синтаксис и семантика.

*Синтаксис*

<променлива> = <израз>;

където

- <променлива> е идентификатор, дефиниран вече като променлива,
- <израз> е израз от тип, съвместим с типа на <променлива>.

*Семантика*

Намира се стойността на <израз>. Ако тя е от тип, различен от типа на <променлива>, конвертира се, ако е възможно, до него и се записва в именуваната с <променлива> памет.

*Забележки:*

• Ако <променлива> е от тип bool, <израз> може да бъде от тип bool или от който и да е числов тип.

• Ако <променлива> е от тип double, всички числови типове, а също типът bool, могат да са типове на <израз>.

• Ако <променлива> е от тип float, типовете float, short, unsiged short и bool, могат да са типове на <израз>. Ако <израз> е от тип int, unsigned int или double, присвояването може да се извърши със загуба на точност. Компилаторът предупреждава за това.

• Ако <променлива> е от тип int, типовете int, long int, short int и bool, могат да са типове на <израз>. В този случай ако <израз> е от тип double или float, дробната част на стойността на <израз> ще бъде отрязана и ако полученото цяло е извън множеството от стойности на типа int, ще се получи случаен резултат. Компилаторът издава предупреждение за това.

• Ако <променлива> е от тип short int, типовете short int и bool, могат да са типове на <израз>. В противен случай се извършват преобразувания, които водят до загуба на точност или даже до случайни резултати. Много компилатори не предупреждават за това.

Ще отбележим, че в рамките на дефиницията на една функция не са възможни две дефиниции на една и съща променлива, но на една и съща променлива може да й бъдат присвоявани многократно различни стойности.

*Пример:* Не са допустими

```
…
double a = 1.5;
…
double a = a + 5.1;
…
```

но са допустими присвояванията:

```
…
double a = 1.5;
…
a = a + 34.5;
…
a = 0.5 + sin(a);
…
```

В езика C++ са въведени някои съкратени форми на оператора за присвояване. Например, заради честото използване на оператора:

a = a + 1;

той съкратено се означава с

a++;

Въведено е също и съкращението a-- на оператора a = a-1;

В същност ++ и -- са реализирани като постфиксни унарни оператори увеличаващи съответно намаляващи аргумента си с 1. Приоритетът им е един и същ с този на унарните оператори +, - и !.

*Забележка:* От оператора ++, за добавяне на 1, идва името на езика C++ - вариант на езика C, към който са добавени много подобрения и нови черти.

*Допълнения:* 1. Операторът за присвояване

<променлива> = <израз>;

съдържа оператора =, реализиран като дясноасоциативен инфиксен аритметично-логически оператор с приоритет по-нисък от този на дизюнкцията ||. Това позволява на <променлива> = <израз>; да се гледа като на израз от тип – типа на <променлива> и стойност – стойността на <израз>, ако е <израз> от типа на <променлива> или стойността на <израз>, преобразувана до типа на <променлива>, ако <променлива> и <израз> не са от един и същ тип.

*Пример:* Програмата

```
#include <iostream.h>
int main()
{int a;
 double  b = 3.2342;
 cout << (a = b) << "\n";
 return 0;
}
```

е допустима. Резултът от изпълнението й е 3, като компилаторът издава предупреждение за загуба на информация при преобразуването от тип double в тип int. Изразът a = b е от тип int и има стойност 3. Ограждането му в скоби е необходимо заради по-ниския приоритет на = от този на <<.

2. Ако е в сила дефиницията:

int x, y, z;

допустим е операторът:

```
x = y = 5;
```

Тъй като = е дясноасоциативен, отначало променливата y се свързва с 5, което е и стойността на израза y = 5. След това x се свързва с 5, което пък е стойността на целия израз.

Някои компилатори издават предупреждение при тази употреба на оператора =. Затова не препоръчваме да се използва = като аритметичен оператор.

## Задачи върху оператора за присвояване

**Задача 12.** Нека са дадени дефинициите

```
double x, y, z;
int m, n, p;
```

Кои от следните редици от символи:

```
а) -x = y;        б) x = -y;        в) m + n = p;
г) p = x + y;     д) z = x - y      е) z = m + n;
ж) sin(0) = 0;    з) x  n + sin(z)  к) 4 = sin(p + 5)
```

са оператори за присвояване, кои не са и защо?

В случаите а), в), ж) и к) редиците не са оператори за присвояване, тъй като се прави опит на израз да се присвои израз. В случай г) редицата от символи е оператор за присвояване, но тъй като на цяла променлива се присвоява стойността на реален израз, компилаторът ще направи предупреждение за загуба на точност, а в случай з) е пропуснат знакът за присвояване. В случаите б), д) и е) редиците са оператори за присвояване.

**Задача 13.** Да се напише програма, която въвежда стойности на реалните променливи a и b, след което разменя и извежда стойностите им (Например, ако a = 5.6, а b = -3.4, след изпълнението на програмата a да става -3.4, а b да получава стойността 5.6).

Програма Zad13.cpp решава задачата.

```
//  Program Zad13.cpp
#include <iostream.h>
```

```
int main()
{cout << "a= ";
 double a;
 cin >> a;
 cout << "b= ";
 double b;
 cin >> b;
 double x; // помощна променлива
 x = a;
 a = b;
 b = x;
 cout << "a= " << a << "\n";
 cout << "b =" << b << "\n";
 return 0;
}
```

В тази програма променливите a и b съдържат входа и изхода от работата на програмата. Използвана е помощна (работна) променлива x, която съхранява първоначалната стойност на променливата a.

**Задача 14.** Да се напише програма, която въвежда положително трицифрено число и извежда на отделни редове цифрите на стотиците, на десетиците и на единиците на числото.

Програма Zad14.cpp решава задачата.

```
// Program Zad14.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{ cout << "a – three-digit, integer and positive? ";
  int a ;
  cin >> a;
  short s, d, e;
  s = a/100;
  d = a/10%10;
  e = a%10;
  cout << setw(10) << "stotici: " << setw(5) << s << "\n";
```

```
  cout << setw(10) << "desetici:" << setw(5) << d << "\n";
  cout << setw(10) << "edinici: " << setw(5) << e << "\n";
  return 0;
 }
```

**Задача 15.** На цялата променлива b да се присвои първата цифра на дробната част на положителното реално число x (Например ако x = 52.467, b = 4).

Програма Zad15.cpp решава задачата

```
// Program Zad15.cpp
#include <iostream.h>
#include <math.h>
int main()
{ cout << "x>0? ";
  double x;
  cin >> x;
  int i = floor(x*10);
  int b = i%10;
  cout << x << "\n";
  cout << b << "\n";
  return 0;
}
```

**Задача 16.** Да се напише програма, която извежда 1, ако в записа на положителното четирицифрено число a, всички цифри са различни и 0 – в противен случай.

Програма Zad16.cpp решава задачата.

```
// Program Zad16.cpp
#include <iostream.h>
int main()
{ cout << "a - four-digit, integer and positive? ";
  int a ;
  cin >> a;
  short h, s, d, e;
```

```
    h = a/1000;
    s = a/100%10;
    d = a/10%10;
    e = a%10;
    cout << (h != s && h != d && h != e &&
             s != d && s != e && d != e)
         << "\n";
    return 0;
  }
```

## 4.2. Празен оператор

Това е най-простия оператор на C++. Описанието му е дадено на Фиг. 4.1.

**Празен оператор**

*Синтаксис*

;

Операторът не съдържа никакви символи. Завършва със знака ;.

*Семантика*

Не извършва никакви действия. Използва се когато синтаксисът на някакъв оператор изисква присъствието на поне един оператор, а логиката на програмата не изисква такъв.

Фиг. 4.1 Празен оператор

*Забележка:* Излишни празни оператори не предизвикват грешка при компилация. Например, редицата от оператори

```
a = 150;;;
b = 50;;;;
c = a + b;;
```

се състои от: оператора за присвояване a = 150;, 2 празни оператора, оператора за присвояване b = 50;, 3 празни оператора, оператора за присвояване c = a + b; и 1 празен оператор и е напълно допустим програмен фрагмент.

Други примери ще дадем по-късно.

## 4.3. Блок

Често синтаксисът на някакъв оператор на езика изисква използването на един оператор, а логиката на задачата – редица от оператори. В този случай се налага оформянето на блок (Фиг. 4.2).

**Блок**

*Синтаксис*

{<оператор$_1$>
 <оператор$_2$>
  . . .
 <оператор$_n$>
}

*Семантика*

Обединява няколко оператора в един, наречен блок. Може да бъде поставен навсякъде, където по синтаксис стои оператор.

Дефинициите в блока, се отнасят само за него, т.е. не могат да се използват извън него.

Фиг. 4.2 Блок

*Пример:* Операторът

```
{cout << “a= “;
 double a;
 cin >> a;
 cout << “b= “;
 double b;
 cin >> b;
 double c = (a+b)/2;
 cout << “average{a, b} = “ << c << “\n”;
}
```

е блок. Опитът за използване на променливите a, b и c след блока, предизвиква грешка.

*Препоръка:* Двойката фигурни скоби, отварящи и затварящи блока да се поставят една под друга.

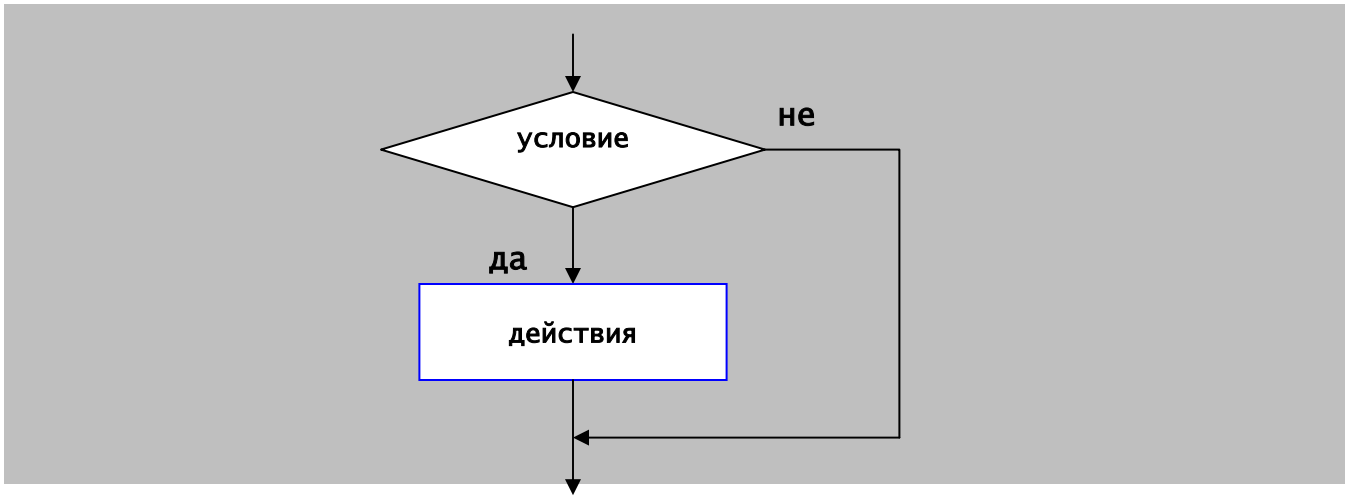
*Забележка:* За разлика от другите оператори, блокът не завършва със знака ;.

## 4.4. Условни оператори

Чрез тези оператори се реализират разклоняващи се изчислителни процеси. Оператор, който дава възможност да се изпълни (или не) един или друг оператор в зависимост от някакво условие, се нарича **условен**. Ще разгледаме следните условни оператори: if, if/else и switch.

### 4.4.1. Условен оператор if

Чрез този условен оператор се реализира разклоняващ се изчислителен процес от вид, илюстриран на Фиг. 4.3.




Фиг. 4.3 Разклоняващ се изчислетелен процес

Ако указано условие е в сила, изпълняват се определени действия, а ако не – тези действия се прескачат. И в двата случая след това се изпълняват общи действия.

Условието се задава чрез някакъв булев израз, а действията – чрез оператор. Фиг. 4.4 описва подробно синтаксиса и семантиката на този оператор.

**Оператор if**

***Синтаксис***

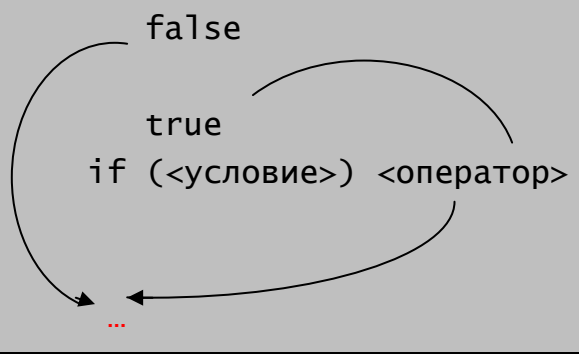
```
if (<условие>) <оператор>
```

където

- if (ако) е запазена дума;
- <условие> е булев израз;
- <оператор> е произволен оператор.

*Семантика*

Пресмята се стойността на булевия израз, представящ условието. Ако резултатът е true, изпълнява се <оператор>. В противен случай <оператор> не се изпълнява, т.е.




Фиг. 4.4 Условен оператор if

*Забележки:*

1. Булевият израз, определящ <условие>, трябва да бъде напълно определен. Огражда се в кръгли скоби.

2. Операторът след условието е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок.

**Задача 17***.* Да се напише програма, която намира най-малкото от три дадени реални числа.

Ще реализираме следните стъпки:

а) Въвеждане на стойности на реалните променливи a, b и c.

б) Инициализиране на работна реална променлива min, която ще играе и ролята на изходна променлива, със стойността на a.

в) Сравняване на b с min. Ако стойността на b е по-малка от запомнения в min текущ минимум, запомня се b в min. В противен случай, min не се променя. Така min съдържа min{a, b}.

г) Сравняване на c с min. Ако стойността на c е по-малка от запомнения в min текущ минимум, c се запомня в min. В противен случай, min не се променя. Така min съдържа min{a, b, c}.

д) Извеждане на резултата – стойността на min.

Програма Zad17.cpp реализира този алгоритъм.

```
// Program Zad17.cpp
#include <iostream.h>
int main()
{cout << "a= ";
 double a;
 cin >> a; // въвеждане на стойност на a
 cout << "b= ";
 double b;
 cin >> b; // въвеждане на стойност на b
 cout << "c= ";
 double c;
 cin >> c; // въвеждане на стойност на c
 double min = a;
 if (b < min) min = b;
 if (c < min) min = c;
 cout << "min{" << a << ", " << b << ", "
      << c << "}= " << min << "\n";
 return 0;
}
```

Ако вместо очаквано реално число, при въвеждане на стойности за променливите a, b и c, се въведе произволен низ, не представляващ число, буферът на клавиатурата, свързан със cin ще изпадне в състояние fail, а обектът cin ще има стойност false. Добрият стил за програмиране изисква в такъв случай програмата да прекъсне изпълнението си с подходящо съобщение за грешка. Програмата от задача 18 реализира този стил.

**Задача 18***.* Да се напише програма, която намира най-малкото от три дадени реални числа. Програмата да извършва проверка за валидност на входните данни.

```
// Program Zad18.cpp
#include <iostream.h>
int main()
{cout << "a= ";
 double a;
 cin >> a;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 cout << "b= ";
 double b;
 cin >> b;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 cout << "c= ";
 double c;
 cin >> c;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 // въведени са валидни стойности за променливите a, b и c
 double min = a;
 if (b < min) min = b;
 if (c < min) min = c;
 cout << "min{" << a << ", " << b << ", "
      << c << "}= " << min << "\n";
 return 0;
}
```

Операторът return предизвиква преустановяване работата на програмата, а указаната в него стойност 1 сигнализира, че е възникнала грешка.

**Задача 19**. Да се сортира във възходящ ред редица от три реални числа, запомнени в променливите a, b и c.

Ще реализираме следните стъпки:

а) Въвеждане на стойности за a, b и c.

б) Сравняване на стойностите на a и b. Ако е в сила релацията b<a, извършва се размяна на стойностите на a и b. В противен случай – размяната не се извършва.

в) Сравняване на стойностите на a и c. Ако е в сила релацията c<a, извършва се размяна на стойностите на a и c. В противен случай – размяната не се извършва. След тези действия, променливата a съдържа най-малката стойност на редицата.

г) Сравняване на стойностите на b и c. Ако е в сила релацията c<b, извършва се размяна на стойностите на b и c. В противен случай – размяната не се извършва.

д) Извеждане на стойностите на a, b, и c.

Програма Zad19.cpp реализира това описание.

```
// Program Zad19.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{ cout << "a= ";
  double a;
  cin >> a;
  if (!cin)
  {cout << "Error, Bad input! \n";
   return 1;
  } // въведена е валидна стойност на a
  cout << "b= ";
  double b;
  cin >> b;
  if (!cin)
  {cout << "Error, Bad input! \n";
   return 1;
  } // въведена е валидна стойност на b
```

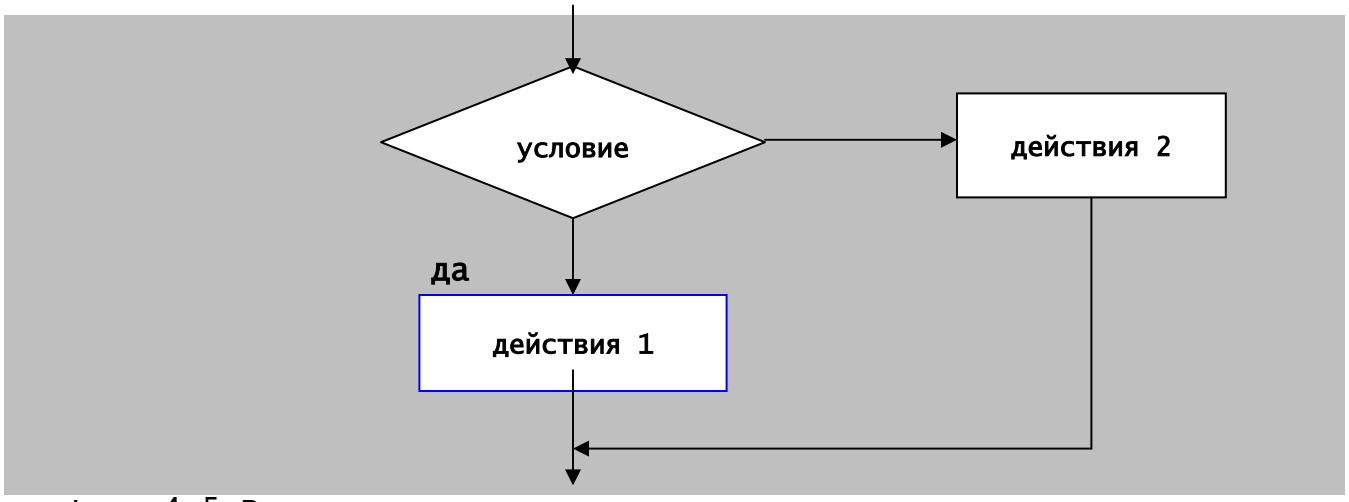
```
    cout << "c= ";
    double c;
    cin >> c;
    if (!cin)
    {cout << "Error. Bad input! \n";
     return 1;
    } // въведена е валидна стойност на c
    if (b < a) {double x = a; a = b; b = x;}
    if (c < a) {double x = c; c = a; a = x;}
    if (c < b) {double x = c; c = b; b = x;}
    cout << setprecision(2) << setiosflags(ios :: fixed);
    cout << setw(10) << a
         << setw(10) << b
         << setw(10) << c << "\n";
    return 0;
  }
```

*Забележка:* Променливата x е видима (може да се използва) само в блоковете, където е дефинирана.

### 4.4.2. Оператор if/else

Операторът се използва за избор на една от две възможни алтернативи в зависимост от стойността на дадено условие.

Чрез него се реализира разклоняващ се изчислителен процес от вид, илюстриран на Фиг. 4.5.




Фиг. 4.5 Разклоняващ се изчислителен процес

Ако указаното условие е в сила, се изпълняват се едни действия, а ако не – други действия. И в двата случая след това се изпълняват общи действия.

Условието се задава чрез някакъв булев израз, а действия 1 и действия 2 – чрез оператори. Фиг. 4.6 описва подробно синтаксиса и семантиката на този оператор.

**Оператор if/else**

*Синтаксис*

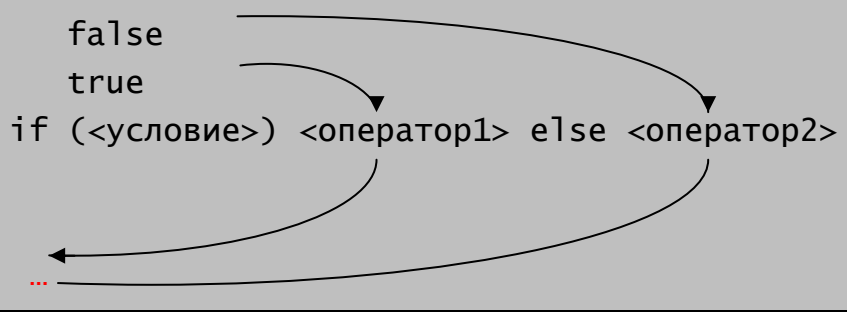
if (<условие>) <оператор1> else <оператор2>

където

- if (ако) и else (иначе) са запазени думи;
- <условие> е булев израз;
- <оператор1> и <оператор2> са произволни оператори.

*Семантика*

Пресмята се стойността на булевия израз, представящ условието. Ако резултатът е true, изпълнява се <оператор1>. В противен случай се изпълнява <оператор2>, т.е.




Фиг. 4.6 Оператор if/else

*Забележки:*

1. Булевият израз, определящ <условие>, трябва да бъде напълно определен. Задължително се огражда в кръгли скоби.

2. Операторът след условието е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок.

3. Операторът след else е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок.

4. Операторът if (<условие>) ; else <оператор> е еквивалентен на оператора if (!<условие>) <оператор>. В него операторът след <условие> е празния.

**Задача 20.** Променливата y зависи от променливата x. Зависимостта е следната:

$$y = \begin{cases} \lg(x) + 1.82, & x \geq 1 \\ x^2 + 7.x + 8.82, & x < 1. \end{cases}$$

Да се напише програма, която по дадено x намира съответната стойност на y.

Програма Zad20.cpp решава задачата.

```
// Program Zad20.cpp
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 } // въведена е валидна стойност за x
 double y;
 if (x >= 1) y = log10(x) + 1.82;
  else y = x*x + 7*x + 8.82;
  cout << setprecision(3) << setiosflags(ios :: fixed);
  cout << setw(10) << x << setw(10) << y << "\n";
  return 0;
}
```

С въвеждането на стойност на променливата x се прави проверка за валидността на въведената стойност. Ако е въведена невалидна стойност, изпълнението на програмата завършва с извеждане на

съобщението “Error! Bad Input!!”. Изпълнението на оператора if/else води до пресмятане на стойността на булевия израз x >= 1. Ако тя е true, се изпълнява операторът за присвояване y = log10(x) + 1.82;. В противен случай се изпълнява операторът за присвояване y = x*x + 7*x + 8.82;, след което се извежда резултатът.

**Вложени условни оператори**

В условните оператори:

if (<условие>) <оператор>

if (<условие>) <оператор1> else <оператор2>

<оператор>, <оператор1> и <оператор2> са произволни оператори, в т. число могат да бъдат условни оператори. В този случай имаме вложени условни оператори.

При влагането е възможно да възникнат двусмислици. Ако в един условен оператор има повече запазени думи if отколкото else, възниква въпросът, за кой от операторите if се отнася съответното else. Например, нека разгледаме оператора

```
if (x >= 0) if ( x >= 5) x = 1/x; else x = -x;
```

Възможни са следните две различни тълкувания на този оператор:

а) if оператор, съдържащ if/else оператор, т.е.

```
if (x >= 0)
   if (x >= 5) x = 1/x; else x = -x;
```

При това тълкуване, ако преди изпълнението на if оператора x има стойност –5, след изпълнението му, стойността на x остава непроменена.

б) if/else оператор, с if оператор след <условие>, т.е.

```
if (x >= 0) if ( x >= 5) x = 1/x;
else x = -x;
```

При това тълкуване, ако преди изпълнението на if/else оператора x има стойност –5, след изпълнението му, стойността на x става 5.

Записът чрез съответни подравнявания, не влияе на компилатора. В езика C++ има правило, което определя начина по който се изпълняват вложени условни оператори.

**Правило:** Всяко else се съчетава в един условен оператор с най-близкото преди него несъчетано if. Текстът се гледа отляво надясно.

Според това правило, компилаторът на C++ ще приеме първото тълкуване за горните вложени условни оператори.

*Препоръка:* Условен оператор да се влага в друг условен оператор само след else. Ако се налага да се вложи след условието, вложеният условен оператор да се направи блок.

### Задачи върху операторите if и if/else

**Задача 21.** Ако променливата a има стойност 8, определете каква стойност ще има променливата b след изпълнението на оператора

```
if (a > 4) b = 5; else
  if (a < 4) b = -5; else
     if (a == 8) b = 8; else b = 3;
```

Тъй като е в сила условието a > 4, променливата b ще получи стойността 5.

**Задача 22.** Стойността на y зависи от x. Зависимостта е следната:

$$y = \begin{cases} x, & x \le 2 \\ 2, & x \in (2, 3] \\ x - 1, & x > 3 \end{cases}$$

Да се напише програма, която по дадено x, намира стойността на y.

Програма Zad22.cpp решава задачата.

```
// Program Zad22.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{ cout << "x= ";
  double x;
  cin >> x;
  if (!cin)
```

```
  { cout << "Error. Bad input! \n";
    return 1;
  }
  double y;
  if (x <= 2) y = x; else
     if (x <= 3) y = 2; else y = x-1;
  cout << setprecision(3) << setiosflags(ios :: fixed);
  cout << setw(10) << x << setw(10) << y << "\n";
  return 0;
 }
```

*Забележка:* В програма Zad22.cpp след първото else е в сила условието x > 2. Затова не е нужно то да се проверява.

**Задача 23***.* Да се напише програма, която въвежда три реални числа a, b и c и извежда 0, ако не съществува триъгълник със страни a, b и c. Ако такъв триъгълник съществува, да извежда 3, 2 или 1 в зависимост от това какъв е триъгълникът – равностранен, равнобедрен или разностранен съответно.

Програма Zad23.cpp решава задачата.

```
// Program Zad23.cpp
#include <iostream.h>
int main()
{cout << "a= ";
 double a;
 cin >> a;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 cout << "b= ";
 double b;
 cin >> b;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
```

```
 }
 cout << "c= ";
 double c;
 cin >> c;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 bool x = a <= 0 || b <= 0 || c <= 0 ||
        a+b <= c || a+c <= b || b+c <= a;
 if (x) cout << 0 << "\n"; else
   if (a == b && b == c) cout << 3 << "\n"; else
     if (a == b || a == c || b == c) cout << 2 << "\n"; else
        cout << 1 << "\n";
 return 0;
}
```

Булевата променлива x е помощна. Тя има стойност true, ако a, b и c не са страни на триъгълник. Получена е след намиране на отрицанието на условието a, b и c да са страни на триъгълник, т.е. на условието

a > 0 && b > 0 && c > 0 && a + b > c && a + c > b && b + c > a

като са използвани законите на де Морган.

**Закони на де Морган:**
!!A е еквивалентно на A
!(A || B) е еквивалентно на !A && !B
!(A && B) е еквивалентно на !A || !B

**Задача 24.** Да се напише програма, която на цялата променлива k присвоява номера на квадранта, в който се намира точка с координати (x, y). Точката не лежи на координатните оси, т.е. x.y ≠ 0.

Програмата Zad24.cpp решава задачата.

```
// Program Zad24.cpp
#include <iostream.h>
int main()
{cout << "x=";
```

```
    double x;
    cin >> x;
    if (!cin)
    {cout << "Error. Bad input! \n";
     return 1;
     }
    cout << "y=";
    double y;
    cin >> y;
    if (!cin)
    {cout << "Error, Bad input! \n";
     return 1;
    }
    if (x*y == 0)
    {cout << "The input is incorrect! \n";
     return 1;
    }
    int k;
    if (x*y>0) {if (x>0) k = 1; else k= 3;}
    else
     if (x>0) k = 4; else k = 2;
    cout << "The point is in: " << k << "\n";
    return 0;
    }
   0
```

### 4.4.3. Оператор switch

Често се налага да се избере за изпълнение един от множество от варианти. Пример за това дава следната задача.

**Задача 25.** Да се напише програма, която въвежда цифра, след което я извежда с думи.

За решаването на задачата трябва да се реализира следната не елементарна функция:

$$f = \begin{cases} \text{zero}, & i = 0 \\ \text{one}, & i = 1 \\ \ldots & \ldots \\ \text{nine}, & i = 9 \end{cases}$$

където с i е означено въведената цифра.

Последното може да стане чрез следната програма:

```
#include <iostream.h>
int main()
{cout << "i= ";
 int i;
 cin >> i;
 if (!cin)
 {cout << "Error, bad input!\n";
  return 1;
 }
 if (i < 0 || i > 9) cout << "Incorrect input! \n";
 else if (i == 0) cout << "zero \n";
 else if (i == 1) cout << "one \n";
 else if (i == 2) cout << "two \n";
 else if (i == 3) cout << "three \n";
 else if (i == 4) cout << "four \n";
 else if (i == 5) cout << "five \n";
 else if (i == 6) cout << "six \n";
 else if (i == 7) cout << "seven \n";
 else if (i == 8) cout << "eight \n";
 else if (i == 9) cout << "nine \n";
 return 0;
}
```

В нея са използвани вложени if и if/else оператори, условията на които сравняват променливата i с цифрите 0, 1, 2, …, 9.

Има по-удобна форма за реализиране на това влагане. Постига се чрез оператора за избор на вариант switch.

Програма Zad25.cpp е друго решение на задачата.

```
// Program Zad25.cpp
#include <iostream.h>
int main()
{cout << "i= ";
 int i;
 cin >> i;
 if (!cin)
```

```
{cout << "Error, bad input!\n";
 return 1;
}
switch (i)
{case 0 : cout << "zero \n"; break;
 case 1 : cout << "one \n";  break;
 case 2 : cout << "two \n";  break;
 case 3 : cout << "three \n"; break;
 case 4 : cout << "four \n"; break;
 case 5 : cout << "five \n"; break;
 case 6 : cout << "six \n"; break;
 case 7 : cout << "seven \n"; break;
 case 8 : cout << "eight \n"; break;
 case 9 : cout << "nine \n"; break;
 default: cout << “Incorrect Input! \n”;
}
return 0;
}
```

Операторът switch започва със запазената дума switch (ключ), следвана от, ограден в кръгли скоби, цял израз. Между фигурните скоби са изброени вариантите на оператора. Описанието им започва със запазената дума case (случай, вариант), следвана в случая от цифра, наречена етикет, двоеточие и редица от оператори.

*Изпълнение на програмата*

След въвеждането на стойност на променливата i се извършва проверка за коректност на въведеното. Нека въведената стойност е 7. Изпълнението на оператора switch причинява да бъде пресметната стойността на израза i – в случая 7. След това последователно сравнява тази стойност със стойностите на етикетите до намиране на етикета 7 и изпълнява редицата от оператори след него. В резултат върху екрана се извежда

seven

курсурът се премества на нов ред и се прекъсва изпълнението на оператора switch. Последното е причинено от оператора break в края на редицата от оператори за варианта с етикет 7.

Операторът switch реализира избор на вариант от множество варианти (възможности). Синтаксисът и семантиката му са дадени на Фиг. 4.7.

**Оператор switch**

*Синтаксис*

switch (<израз>)

{case <израз$_1$> : <редица_от_оператори$_1$>

 case <израз$_2$> : <редица_от_оператори$_2$>

 …

 case <израз$_{n-1}$> : <редица_от_оператори$_{n-1}$>

 [default : <редица_от_оператори$_n$>]$_{опц}$

}

където

- switch (ключ), case (случай, избор или вариант) и default (по премълчаване) са запазени думи на езика;
- <израз> е израз от допустим тип (Типовете bool, int и char са допустими, реалните типове double и float не са допустими). Ще го наричаме още switch-израз.
- <израз$_1$>, <израз$_2$>, …, <израз$_{n-1}$> са константни изрази, задължително с различни стойности.
- <редица_от_оператори$_i$> (i = 1, 2, …, n) се дефинира по следния начин:

<редица_от_оператори> ::= <празно>|
<оператор>|
<оператор><редица_от_оператори>

*Семантика*

Намира се стойността на switch-израза. Получената константа се сравнява последователно със стойностите на етикетите <израз$_1$>, <израз$_2$>, … При съвпадение, се изпълняват операторите на съответния вариант и операторите на всички варианти, разположени след него, до срещане на оператор break. В противен случай, ако участва default-вариант, се изпълнява редицата от оператори, която му съответства и редиците от оператори на всички варианти след него до достигане до break и в случай, че не участва такъв – не следват никакви действия от оператора switch.

Фиг. 4.7 Оператор switch

Между фигурните скоби са изброени вариантите на оператора. Всеки вариант (без евентуално един) започва със запазената дума case, следвана от израз (нарича се още case–израз или етикет), който трябва да може да се пресметне по време на компилация. Такива изрази се наричат **константни**. Те не зависят от входните данни. След константния израз се поставя знакът двоеточие, следван от редица от оператори (оператори на варианта), която може да е празна. Сред вариантите може да има един (не е задължителен), който няма case-израз и започва със запазената дума default. Той се изпълнява в случай, че никой от останалите варианти не е бил изпълнен.

*Забележка:* Не е задължително default – вариантът (ако го има) да е последен, но добрият стил за програмиране го изисква.

Съществува възможност програмистът да съобщи на компилатора, че желае да се изпълни само редицата от оператори на варианта с етикет, съвпадащ със стойността на switch-израза, а не и всички следващи го. Това се реализира чрез използване на оператор break в края на редицата от оператори на варианта. Този оператор предизвиква прекъсване на изпълнението на оператора switch и предаване на управлението на първия оператор след него (Фиг. 4.8.).

Операторът break принадлежи към групата на т. нар. **оператори за преход**. Тези оператори предават управлението безусловно в някаква точка на програмата.

**Оператор break**

*Синтаксис*

break;

*Семантика*

Прекратява изпълнеието на най-вътрешния съдържащ го оператор switch или оператор за цикъл. Изпълнението на програмата продължава от оператора, следващ (съдържащ) прекъснатия.

Фиг. 4.8 Оператор break

Програмистът съзнателно пропуска оператора break, когато за няколко различни стойности от множеството от стойности, трябва да се извършат еднакви действия.

*Пример:* Следният програмен фрагмент извежда дали въведена цифра е четно или нечетно число.

```
…
switch (i)
{case 1:
 case 3:
 case 5:
 case 7:
 case 9: cout << "odd number \n"; break;
 case 0:
 case 2:
 case 4:
 case 6:
 case 8: cout << "even number \n";
}
```

*Забележка:* Ако във вариантите на оператора switch не е използван операторът break, ще бъде изпълнена редицата от оператори на варианта, чийто case-израз съвпада със стойността на switch-израза и също всички след нея.

Използването на оператора switch има едно единствено предимство пред операторите if и if/else – прави реализацията по-ясна. Основен негов недостатък е, че може да се прилага при много специални обстоятелства, произтичащи от наложените ограничения на типа на switch-израза, а именно, той трябва да е цял, булев или символен. Освен това, използването на оператора break, затруднява доказването на важни математически свойства на програмите.

## Задачи върху оператора switch

**Задача 26.** Да се напише програма, която по зададено реално число x намира стойността на един от следните изрази:

y = x - 5
y = sin(x)
y = cos(x)
y = exp(x).

Изборът на желания израз да става по следния начин: при въвеждане на цифрата 1 се избира първият, на 2 – вторият, на 3 – третият и на 4 – четвъртият израз.

Програма Zad25.cpp решава задачата.

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
{cout << "================================\n";
 cout << "|    y = x-5            -> 1    |\n";
 cout << "|    y = sin(x)         -> 2    |\n ";
 cout << "|    y = cos(x)         -> 3    |\n";
 cout << "|    y = exp(x)         -> 4    |\n";
 cout << "================================\n";
 cout << " 1, 2, 3 or 4? \n";
 int i;
 cin >> i;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (i == 1 || i == 2 || i == 3 || i == 4)
 {cout << "x= ";
  double x;
  cin >> x;
  if (!cin)
  {cout << "Error. Bad input! \n";
   return 1;
  }
  double y;
  switch (i)
```

```
     {case 1: y = x - 5;  break;
      case 2: y = sin(x); break;
      case 3: y = cos(x); break;
      case 4: y = exp(x); break;
     }
     cout << "y= " << y << "\n";
    }
    else
    {cout << "Error. Bad choice! \n";
     return 1;
    }
    return 0;
   }
```

## 4.5. Оператори за цикъл

Операторите за цикъл се използват за реализиране на циклични изчислителни процеси.

Изчислителен процес, при който оператор или група оператори се изпълняват многократно за различни стойности на техни параметри, се нарича **цикличен**.

Съществуват два вида циклични процеси:

- индуктивни и
- итеративни.

Цикличен изчислителен процес, при който броят на повторенията е известен предварително, се нарича **индуктивен цикличен процес**.

*Пример:* По дадени цяло число n и реално число x, да се намери сумата

$$S = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}.$$

Ако S има начална стойност 1, за да се намери сумата е необходимо n пъти да се повторят следните действия:

а) конструиране на събираемо

$$\frac{x^i}{i!}, \ (i = 1, 2, \ldots, n)$$

# 5

# Скаларни типове символен и изброен

## 5.1 Тип символен

Досега използвахме главно числови данни. В редица приложения се налага да се описва и обработва и символна информация. Да разгледаме следната задача.

**Задача 44.** Да се напише програма, която намира стойността на числов израз без скоби и без приоритет на операторите (+, -, * и /). Например, стойността на израза 12-3*2/4 е 4.5, а не 10.5. Пресмятането завършва след въвеждане на знака =.

Тази задача изисква програмата да получава за вход редица от числа, знаците за аритметични операции +, -, * и /, да завършва със знака = и да намира стойността на въведения аритметичен израз.

За реализирането й ще използваме данни от тип символен. Ще се върнем към задачата след разглеждането на този тип.

Типът се нарича още **знаков** и се причислява към интегралните типове на езика. За означаването му се използва запазената дума char.

**Множество от стойности**

Състои се от крайно и наредено множество от символи, обикновено вариант на ISO-646, например ASCII, и обхваща символите от клавиатурата. Стандартизирана е само част от символното множество – 128 символа (главните и малки буква на латинската азбука, цифрите, символите за пунктуация и някои управляващи символи). Всяка реализация включва стандартизираното подмножество и има свободата на избор за останалите символи, което създава редица проблеми.

Реализацията Visual C++ 6.0 включва в множеството от стойностите си два вида символи – **графични** и **управляващи**.

Графичните символи имат видимо представяне. Означават се чрез ограждане на символа в апострофи.

*Пример:* ‘a’ ‘d’ ‘K’ ‘!’ ‘1’ ‘ ‘ ‘+’

са означения на символите a, d, K, !, 1, интервал и +.

Графичните символи \ и ‘ се представят чрез ‘\\’ и ‘\’’ съответно. Символът ? се представя и като ‘?’ ,и като ‘\?’.

Управляващите символи нямат видимо представяне. Означават се по следния начин:

‘\символ’

Фиг. 5.1 представя някои от тези символи, а също и предназначението им.

| Символ | Предназначение |
|---|---|
| \a | издава звуков сигнал |
| \b | връща курсора един символ назад |
| \n | предизвиква преминаване на нов ред |
| \r | връща курсора в началото на реда |
| \t | хоризонтална табулация |
| \v | вертикална табулация |
| \0 | нулев символ, край на низ |

Фиг. 5.1 Някои управляващи символи

Елементите от множеството от стойности на типа символен са константите на този тип. Наричат се още **символни литерали**.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа символен, се нарича

**символна променлива** или **променлива от тип символен**. Дефинира се по общоприетия начин.

*Примери:*

```
char c1;
char c2 = '+';
```

Дефиницията свързва променливите с множеството от стойности на типа char или с конкретна стойност от това множество като отделя по 1 байт оперативна памет за всяка от тях. Стойността на тази памет е неопределена или е константата, свързана с дефинираната променлива, в случай, че дефиницията е с инициализация.

След дефиницията от примера по-горе, имаме:

```
ОП
c1      c2
_       +
1в      1в
```

Стойността на клетките от паметта, именувани със c1 е неопределена, а това на именуваните със c2 е '+'. В същност, вместо '+' в паметта е записан кодът (вътрешното представяне) на символът '+' – цялото число 43.

Елементите от множеството от стойности на типа char е наредено. Всеки символ е свързан с код, съгласно кодирането ASCII. При това кодиране, за управляващите символи се използват целите числа от 0 до 31 включително. Останалите символи се кодират с числата от 32 до 255. Фиг. 5.2 дава частична представа за наредбата на някои символи.

**ASCII-кодове на някои символи**

| 0 | 31 | 32 | 48 | 49 … 57 | … | 65 | 66 … 90 | … | 97 | 98 … 122 |
|---|---|---|---|---|---|---|---|---|---|---|
| управляващи символи | интервал | | 0 | 1 … 9 | … | A | B … Z | … | a | b … z |

Фиг. 5.2 ASCII-кодове на някои символи

**Забележка:** Цифрите, а също главните и малки букви на латинската азбука имат последователни кодове.

## Операции над символни данни

### *Намиране на кода на символ*

Извършва се чрез израза (int)c1, където c1 е символна константа или променлива.

*Пример:* Операторът

cout << (int)'F';

извежда кода на главната латинска буква F.

### *Намиране на символ по даден код*

Осъществява се чрез израза (char)<цял_аритметичен_израз>. Стойността на <цял_аритметичен_израз> трябва да е неотрицателно цяло число. Ако не е от интервала [0, 255], намира се остатъкът от делението по модул 256.

*Пример:* Операторът

cout << (char)65 << '\t' << (char)(3*256+65);

извежда два пъти символа A главно, латинско, разделени с табулация.

### *Аритметични операции*

Всички аритметични операции, допустими над целочислени данни са допустими и за данни от тип char. Извършват се над кодовете им. Резултатът е цяло число. Така се разширява синтаксисът на целите изрази.

*Примери:*

1. c1 + 15 - 'A' е цял аритметичен израз, стойността на който се получава като се събере кодът на символа, свързан със c1, с 15 и от полученото цяло число се извади 65 (кодът на 'A').

2. c1%2 е цял аритметичен израз.

### *Присвояване на стойност*

На променлива от символен тип може да се присвояват символни константи и променливи, а също стойностите на цели аритметични изрази. В последния случай, добрият стил на програмиране изисква да се конвертира явно типът на целия аритметичен израз до символен.

*Примери:*

```
c1 = 'A'
c2 = c1;
c1 = c1 + c2 - 15;
```

В третия пример компилаторът автоматично ще извърши преобразуването от тип цял в тип char. По-добре е явно това да се укаже, т.е. c1 = (char)(c1 + c2 – 15); е по-добър вариант.

### *Намиране на символа пред и след указан символ*

Символът, пред символа c е (char)(c-1), а този след c е (char)(c+1), където c е константа или променлива от символен тип.

**Забележка:** Ако c е променлива от символен тип, това може да се постигне и чрез c++ и c--, но се променя стойността на c.

### *Логически операции*

Всички логически операции са допустими и над данни от символен тип. Изпълняват се над кодовете им. Резултатът е от булев тип. Освен това, символна константа или променлива, поставена на място на условие, изпълнява ролята на булев израз.

### *Операции за сравнение*

Над данни от тип символен могат да се прилагат стандартните инфиксни оператори за сравнение:

| Оператор | Операция |
|---|---|
| == | сравнение за равно |
| != | сравнение за различно |
| > | сравнение за по-голямо |
| >= | сравнение за по-голямо или равно |

| | |
|---|---|
| < | сравнение за по-малко |
| <= | сравнение за по-малко или равно |

Сравняват се кодовете. Резултатът е булев.

*Примери:*

‘a’ <= ‘z’  е true  ‘a’ <= ‘0’ е false

‘0’ < ‘9’  е true  ‘A’ > ‘a’ е false

‘x’ != ‘X’  е true

Данни от символен тип могат да се сравняват чрез горните оператори и с цели аритметични изрази. Резултатът е булев.

*Примери:*

‘a’ == 0  c1 != 65  c2 >= 122

Така се разширява синтаксисът на булевите изрази.

***Въвеждане***

Осъществява се чрез оператора >>. В този случай, операторът >> не е чувствителен към символите: интервал, хоризонтална и вертикална табулации и преминаване на нов ред.

*Примери:*

1. Операторът

cin >> c1 >> c2;

се изпълнява по следния начин: Ако в буфера на клавиатурата няма символи, различни от интервали, табулации и знаци за преминаване на нов ред, настъпва пауза до въвеждане на два символа, различни от интервали, табулации и знаци за преминаване на нов ред и разделени със същите знаци. Те обслужват последователно c1 и c2. В противен случай, не настъпва пауза, а символите от буфера се свързват последователно със c1 и c2.

1. Програмният фрагмент

```
cout << “c1= “;
char c1;
cin >> c1;
cout << “c2= “;
char c2;
cin >> c2;
```

се изпълнява по следния начин: Върху екрана се появява подсещането c1=. Операторът cin >> c1; очаква въвеждане на един символ, различен от интервал, табулация и знак за преминаване на нов ред. Въвеждането трябва да завършва със знака за преминаване на нов ред. Водещите интервали, табулации и знаци за преминаване на нов ред се пренебрегват. След това се повтарят същите действия за променливата c2 освен ако след подсещането за стойност на c1 не са въведени два символа (може да са без разделител по между си), различни от интервали, табулации и знаци за преминаване на нов ред. Тогава първият символ се свързва с променливата c1, а вторият – с променливата c2.

***Извеждане***

Осъществява се по стандартния начин.

*Пример:* Операторът

```
cout << c1;
```

ще изведе символът, свързан със c1, а операторът

```
cout << c1+c2;
```

ще изведе цялото число, равно на сумата от кодовете на c1 и c2, тъй като c1+c2 е аритметичен израз. Ако се налага да се изведе символът, чийто код е равен на сумата от кодовете на c1 и c1, трябва да се използва операторът

```
cout << char(c1+c2);
```

**Допълнения:**

1. Типът char е допустим за тип на switch-израз, т.е. допустим е фрагментът:

```
char c;
cin >> c;
switch (c)
{case ‘a’: …
 case ‘e’: …
 …
}
```

2. Символните константи могат да съдържат и повече от един символ, например ‘ah’, ‘HAHA’ са символни константи. Тази възможност няма да бъде застъпена в нашия курс.

3. Чрез модификаторите signed и unsigned се получават разновидности на типа char. Те също няма да бъдат застъпени, тъй като реализацията Visual C++ 6.0 не е чувствителна към тях.

## Задачи върху тип символен

Нека се върнем към задача 44. Програма Zad44.cpp дава едно нейно решение.

```
// Program Zad44.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{char op = '+';
 double result = 0.0;
 do
 {double arg;
  cin >> arg;
  switch (op)
  {case '+': result = result + arg; break;
   case '-': result = result - arg; break;
   case '*': result = result * arg; break;
   case '/': result = result / arg;
  }
  cin >> op;
 } while (op != '=');
 cout << setprecision(5) << setiosflags(ios::fixed);
 cout << setw(15) << result << "\n";
 return 0;
}
```

**Допълнение:** Операторът

```
switch (op)
```

```
     {case '+': result = result + arg; break;
      case '-': result = result - arg; break;
      case '*': result = result * arg; break;
      case '/': result = result / arg;k;
     }
```

може да се запише и в следната съкратена форма:

```
   switch (op)
     {case '+': result += arg; break;
      case '-': result -= arg; break;
      case '*': result *= arg; break;
      case '/': result /= arg;
     }
```

**Задача 45.** Да се напише програма, която въвежда малка буква от латинската азбука и извежда съответната й главна буква.

Програма Zad45.cpp решава задачата.

```
// Program Zad45.cpp
#include <iostream.h>
int main()
{char c;
 cin >> c;
 if (c < 'a' || c > 'z')
 {cout << "Incorrect Input! \n";
  return 1;
 }
 cout << (char)(c - 'a' + 'A') << '\n';
 return 0;
}
```

**Задача 46.** Да се напише програма, която извежда цифрите, главните и малките букви на латинската азбука и ASCII кодовете им. Всеки символ и кодът му да са на един и същ ред. За разделител между символ и код да служи знакът за хоризонтална табулация.

Програма Zad46.cpp решава задачата.

```
// Program Zad46.cpp
#include <iostream.h>
int main()
{for(char ch = '0'; ch <= '9'; ch++)
 cout << ch << '\t' << (int)ch << '\n';
 for(ch = 'A'; ch <= 'Z'; ch++)
 cout << ch << '\t' << (int)ch << '\n';
 for(ch = 'a'; ch <= 'z'; ch++)
 cout << ch << '\t' << (int)ch << '\n';
 return 0;
}
```

## 5.2 Тип изброен

Типът е стандартен. За разлика от другите, досега разгледани типиве, той се дефинира от програмиста като се изброяват константите му. Затова се нарича още потребителски дефиниран тип.

**Дефиниция на тип изброен**

Фиг. 5.3 илюстрира синтаксиса на дефиницията му.

**Дефиниране на тип изброен**

<дефиниция_на_тип_изброен> ::=
enum [<име_на_тип>]$_{опц}$ {<идентификатор$_1$> [= <константен_израз$_1$>]$_{опц}$,
<идентификатор$_2$> [= <константен_израз$_2$>]$_{опц}$,
…
<идентификатор$_n$> [= <константен_израз$_n$>]$_{опц}$};
<име_на_тип> ::= <идентификатор>

където

- enum е запазена дума (съкращение от enumerate – изброявам);
- <константен_израз$_i$> (i = 1, 2,…, n) е константен израз от интегрален тип.

Фиг. 5.3 Дефиниране на тип изброен

Името на типа, а също <константен_израз$_i$> (i = 1, 2, …, n) могат да се пропуснат. Пропускането на <константен_израз$_i$> става заедно със знака =. Изброените идентификатори трябва да са различни не само в рамките на една дефиниция, а и в тези на всички дефиниции на изброени типове, в рамките на даден модул.

*Примери:*

```
enum Weekday{SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
             THURSDAY, FRIDAY, SATURDAY};
enum Name{IVAN=5, PETER=3, MERY=8, SONIA=6, VERA=10};
enum Id{A1, A2, A3, A4=8, A5, A6=10, A7, A8};
enum {FALSE, TRUE};
```

Забелязваме, че в четвъртата дефиниция на примера е пропуснато името на типа. Такива типове се наричат анонимни изброени типове.

Типът изброен дава един начин за дефиниране на константи.

**Множество от стойности**

Състои се от всички изброени в дефиницията на типа идентификатори.

*Пример:*

множество от стойности на тип Weekday

Елементите от множеството от стойности на даден изброен тип са константите на този тип. Затова ще ги означаваме с главни букви.

*Примери:* SATURDAY е константа от тип Weekday, VERA е константа от тип Name. TRUE и FALSE също са константи, но от безименен тип изброен.

Променлива величина, приемаща за стойности константи от множеството от стойности на някакъв изброен тип, се нарича **променлива от този тип изброен**. Дефинира се по общоприетия начин.

*Примери:*

```
Weekday d1, d2 = SUNDAY;
Name a;
Name b = VERA;
Id x = A2, y = A7;
```

Тук d1 и d2 са променливи от тип Weekday, а и b – от тип Name, а x и y – от тип Id.

Дефиницията свързва променливите от даден тип изброен с множеството от стойности на типа или с конкретна стойност от това множество като отделя по 1 или 4 байта ОП за всяка от тях (за реализацията Visual C++ 6.0 - 4 байта ОП). Стойността на тази памет е неопределена или е константата, свързана с дефинираната променлива, в случай, че тя е инициализирана.

След дефиницията от примера по-горе, имаме:

ОП

| d1 | d2 | a | b | x | y |
|---|---|---|---|---|---|
| - | SUNDAY | - | VERA | A2 | A7 |
| 4 байта | 4 байта | 4 байта | 4 байта | 4 байта | 4 байта |

Стойността на паметта, именувана с d1 и a, е неопределена, това наименуваната с d2 е SUNDAY, на b – VERA, на x и y – A2 и A7 съответно.

В същност, вместо SUNDAY, VERA, A2 и A7, в паметта са записани кодовете им (вътрешните им представяния).

Вътрешните представяния на константите от изброени типове се определят по следния начин:

- Ако не са указани стойности, по подразбиране първият идентификатор получава стойност 0, а всеки следващ – стойност с единица по-голяма от стойността на предходния.

*Пример:* Вътрешните представяния на константите от тип Weekday са:

SUNDAY - 0, MONDAY - 1, TUESDAY – 2 и т.н. SATURDAY – 6.

- Ако всички идентификатори са свързани със стойности, вътрешното представяне на всяка константа от такъв изброен тип е указаната стойност.

*Пример:* Вътрешните представяния на константите от тип Name са:

IVAN - 5, PETER - 3, MARY - 8, SONIA - 6, VERA - 10.

• Ако някои идентификатори са свързани със стойности, а други – не, вътрешното представяне на идентификатор, за който е указана стойност е указаната стойност, а на всеки идентификатор, за който не е указана стойност – е вътрешното представяне на идентификатора пред него, увеличено с 1. Вътрешното представяне на първият идентификатор, ако не е указана стойност за него, е 0.

*Пример:* Вътрешните представяния на константите от тип Id са:
A1 - 0, A2 - 1, A3 - 2, A4 - 8, A5 – 9, A6 – 10, A7 = 11 и A8 – 12.

Възможни са и дефиниции на променливи от тип изброен от вида:

enum {RED, BLUE = 4, WHITE, BLACK = 7} color1, color2;

Тази дефиниция показва, че вместо име на тип, може да се използва *дефиниция* на анонимен изброен тип.

**Операции върху данни от тип изброен**

***Намиране на кода на константа от тип изброен***

Извършва се чрез израза (int)x, където x е константа или променлива от изброен тип.

*Пример:* Операторът

cout << (int)FRIDAY;

извежда 5 - кода на FRIDAY.

***Намиране на константа от тип изброен по даден код***

Осъществява се чрез израза (<име_на_тип_изброен>)<код>.

*Пример:* Изразът d1 = (Weekday)4 намира THURSDAY.

***Аритметични операции***

Всички аритметични операции, допустими над целочислени данни са допустими и за данни от тип изброен. Извършват се над кодовете на данните. Резултатът е цяло число.

*Примери:*

1. d1+d2-4 е цял аритметичен израз, стойността на който се получава като се съберат кодовете на d1 и d2 и от полученото се извади 4.

2. d1 % 5 е цял аритметичен израз, изразяващ остатъка от делението на кода на d1 на 5.

*Присвояване на стойност*

На променлива от тип изброен може да се присвои която и да е константа от множеството от стойности на типа, от който е променливата, а също и стойността на променлива от същия изброен тип.

*Пример:*

```
d1 = WEDNESDAY;
d2 = d1;
```

Присвояването

```
d1 = d2 - 2;
```

не е допустимо, но

```
d1 = (Weekday)(d2 - 2);
```

вече е допустимо.

*Логически операции*

Всички логически операции, са допустими и за данни от тип изброен. Извършват се на кодовете на данните. Резултатът е булев.

*Операции за сравнение*

Данни от един и същ тип изброен могат да се сравняват чрез стандартните инфиксни оператори за сравнение:

| Оператор | Операция |
|---|---|
| == | сравнение за равно |
| != | сравнение за различно |
| > | сравнение за по-голямо |
| >= | сравнение за по-голямо или равно |
| < | сравнение за по-малко |

| | |
|---|---|
| <= | сравнение за по-малко или равно |

Сравняват се кодовете. Резултатът е булев.

*Примери:*

SUNDAY <= FRIDAY  е true

VERA < PETER е false

A3 != A7  е true

***Въвеждане***

Не е възможно въвеждане на стойност на променлива от някакъв изброен тип чрез оператора >>, т.е. операторът

cin >> d1;

е недопустим.

***Извеждане***

Операторите

cout << <константа_от_тип_изброен>;

или

cout << <променлива_от_тип_изброен>;

извеждат кода на константата или променливата.

Следният програмен фрагмент дефинира изброен тип с име Weekday и извежда стойността на променлива от този тип.

```
…
enum Weekday{SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
             THURSDAY, FRIDAY, SATURDAY};
Weekday d;
d = FRIDAY;
switch(d)
{case SUNDAY: cout << "SUNDAY \n"; break;
 case MONDAY: cout << "MONDAY \n"; break;
 case TUESDAY: cout << "TUESDAY \n"; break;
 case WEDNESDAY: cout << "WEDNESDAY \n"; break;
 case THURSDAY: cout << "THURSDAY \n"; break;
 case FRIDAY : cout << "FRIDAY \n"; break;
```

```
 case SATURDAY: cout << "SATURDAY \n";
}
```

Типът изброен е потребителски дефиниран тип. Заради това, всеки потребител може да предефинира операторите ++, --, << и др. за работа с данни от дефиниран от него изброен тип.

**Задача 47.** Да се напише програма, която намира средната седмична температура.

Програма Zad47.cpp решава задачата.

```
// Program Zad47.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{enum Weekday{SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
              THURSDAY, FRIDAY, SATURDAY};
 Weekday d;
 double s = 0.0;
 for(d = SUNDAY; d <= SATURDAY; d = (Weekday)(d+1))
 {double t;
  cout << "t= ";
  cin >> t;
  s = s + t;
 }
 cout << setprecision(2) << setiosflags(ios :: fixed);
 cout << setw(10) << s/7 << "\n";
 return 0;
}
```

Забележете частта <корекция> на оператора for. Операторът за просвояване d++ не е допустим за коректор. Тъй като вътрешното представяне на d е цяло число, възможно е да се увеличи с 1, т.е. d+1 е допустимо, но присвояването му на променливата d от тип Weekday не е възможно. Налага се преобразуване на цялото число d+1 в тип Weekday.

Изброеният тип задава крайно множество от константи. Единствената разлика между константите от тип изброен и еквивалентните им const – дефиниции е, че с тях не е свързан адрес от паметта.

Основното предимство от използването на тип изброен е подобряването читаемостта на програмата. Това се счита за добър стил за програмиране.

## Задачи

**Задача 1.** Да се напише програма, която извежда върху екрана следната таблица:

```
а) A B C D E              б) A
   B C D E F                 B C
   C D E F G                 C D E
   D E F G H                 D E F G
   E F G H I                 E F G H I
```

**Задача 2.** Да се напише програма, която въвежда стойности на естествените числа n, m, p и q и намира всички редици от операции (op1, op2, op3) сред +, -, * и /, така че като се зместят в израза (((n op1 m) op2 p) op3 q), получената стойност да е равна на a (a дадено цяло число).

**Задача 3.** Да се напише програма, която въвежда стойности на естествените числа n, m, p и q и установява дали съществува редица от операции (op1, op2, op3) сред +, -, * и /, така че като се зместят в израза (((n op1 m) op2 p) op3 q), получената стойност да е равна на a (a дадено цяло число).

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. М. Тодорова, Програмиране на Паскал, Полипринт, София, 1993.

# 6

# Съставни типове данни. Масив. Символен низ

## 6.1 Структура от данни масив

Под структура от данни се разбира **организирана информация, която може да бъде описана, създадена и обработена с помощта на програма.**

За да се определи една структура от данни е необходимо да се направи:

- **логическо описание на структурата**, което я описва на базата на декомпозицията й на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции.
- **физическо представяне на структурата**, което дава методи за представяне на структурата в паметта на компютъра.

В предходните глави разгледахме структурите числа и символи. За всяка от тях в езика C++ са дадени съответни типове данни, които ги реализират. Тъй като елементите на тези структури се състоят от една компонента, те се наричат **прости**, или **скаларни**.

Структури от данни, компонентите на които са редици от елементи, се наричат **съставни**.

Структури от данни, за които операциите включване и изключване на елемент не са допустими, се наричат **статични**, в противен случай - **динамични**.

В тази глава ще разгледаме структурата от данни масив и средствата, които я реализират.

**Логическо описание**

Масивът е крайна редица от фиксиран брой елементи от един и същ тип. Към всеки елемент от редицата е възможен пряк достъп, който се осъществява чрез индекс. Операциите включване и изключване на елемент в/от масива са недопустими, т.е. масивът е статична структура от данни.

**Физическо представяне**

Елементите на масива се записват последователно в паметта на компютъра, като за всеки елемент на редицата се отделя определено количество памет.

В езика C++ структурата масив се реализира чрез типа масив.

## 6.2 Тип масив

В C++ структурата от данни масив е реализирана малко ограничено. Разглежда се като крайна редица от елементи от един и същ тип с пряк достъп до всеки елемент, осъществяващ се чрез индекс с цели стойности, започващи от 0 и нарастващи с 1 до указана горна граница. Дефинира се от програмиста.

**Дефиниране на масив**

Типът масив се определя чрез задаване на типа и броя на елементите на редицата, определяща масив. Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален. За типа T и константния израз от интегрален или изброен тип с положителна стойност size, T[size] е тип масив от size елемента от тип T. Елементите се индексират от 0 до size-1. T се нарича **базов тип** за типа масив, а size – горна граница.

*Примери:*

int[5] дефинира масив от 5 елемента от тип int, индексирани от 0 до 4;

double[10] дефинира масив от 10 елемента от тип double, индексирани от 0 до 9;

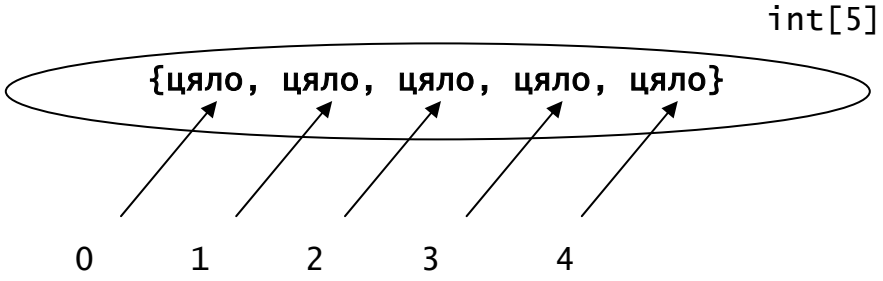
bool[4] дефинира масив от 4 елемента от тип bool, индексирани от 0 до 3.

**Множество от стойности**

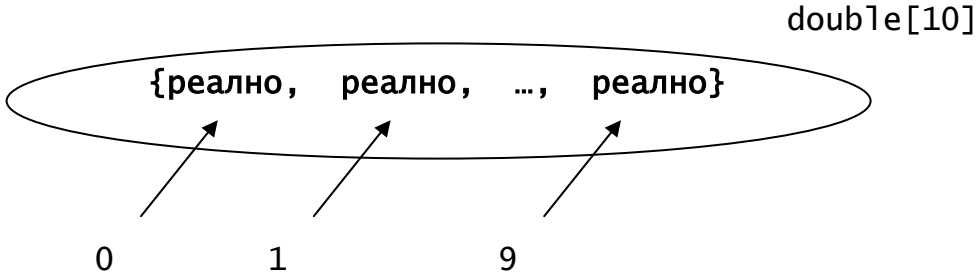
Множеството от стойности на типа T[size] се състои от всички редици от по size елемента, които са произволни константи от тип T. Достъпът до елементите на редиците е пряк и се осъществява с помощта на индекс, като достъпът до първия елемент се осъществява с индекс със стойност 0, до последния – с индекс със стойност size-1, а до всеки от останалите елементи – с индекс със стойност с 1 по-голяма от тази на индекса на предишния елемент.

*Примери:*

1. Множеството от стойности на типа int[5] се състои от всички редици от по 5 цели числа. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2, 3 и 4.




2. Множеството от стойности на типа double[10] се състои от всички редици от по 10 реални числа. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2, 3 и т.н. 9.




Елементите от множеството от стойности на даден тип масив са **константите** на този тип масив.

*Примери:*

1. Следните редици {1,2,3,4,5}, {-3, 0, 1, 2, 0}, {12, -14, 8, 23, 1000} са константи от тип int[5].

2. Редиците {1.5, -2.3, 3.4, 4.9, 5.0, -11.6, -123.56, 13.7, -32.12, 0.98}, {-13, 0.5, 11.9, 21.98, 0.03, 1e2, -134.9, 0.09, 12.3, 15.6} са константи от тип double[10].

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип масив, се нарича променлива от дадения тип масив. Понякога ще я наричаме само масив.

Фиг. 6.1 определя дефиницията на променлива от тип масив. Тук общоприетият запис е нарушен. Променливата се записва между името на типа и размерността.

**Дефиниция на масив**

<дефиниция_на_променлива_от_тип_масив> ::=

T <променлива>[size] [= {<редица_от_константни_изрази>}]$_{опц}$

{,<променлива>[size] [= {<редица_от_константни_изрази>}]$_{опц}$ }$_{опц}$;

където

- T е име или дефиниция на произволен тип, различен от псевдоним, void, функционален;
- <променлива> е идентификатор;
- size е константен израз от интегрален или изброен тип с *положителна* стойност;
- <редица_от_константни_изрази> се дефинира по следния начин:

<редица_от_константни_изрази> ::= <константен_израз>|

<константен_израз>, <редица_от_константни_изрази>

като константните изрази са от тип T или от тип, съвместим с него.

Фиг. 6.1 Дефиниция на масив

*Примери:*

```
int a[5];
double c[10];
```

```
bool b[3];
enum {FALSE, TRUE} x[20];
double p[4] = {1.25, 2.5, 9.25, 4.12};
```

Дефиницията

```
T <променлива>[size] = {<редица_от_константни_изрази>}
```

се нарича **дефиниция на масив с инициализация**, а фрагмента {<редица_от_константни_изрази>} - **инициализация**. При нея е възможно size да се пропусне. Тогава за стойност на size се подразбира броят на константните изрази, изброени в инициализацията. Ако size е указано и изброените константни изрази в инициализацията са по-малко от size, останалите се приемат за 0.

*Примери:*

1. Дефиницията

```
int q[5] =  {1, 2, 3};
```

е еквивалентна на

```
int q[] =  {1, 2, 3, 0, 0};
```

2. Дефиницията

```
double r[] = {0, 1, 2, 3};
```

е еквивалентна на

```
double r[4] = {0, 1, 2, 3};
```

**Забележка:** Не са възможни конструкции от вида:

```
int q[5];
q = {0, 1, 2, 3, 4};
```

а също

```
int q[];
```

и

```
double r[4] = {0.5, 1.2, 2.4, 1.2, 3.4};
```

Ще отбележим, че фрагментите

```
<променлива>[size] и
<променлива>[size] = {<редица_от_константни_изрази>}
```

от дефиницията от Фиг. 6.1 могат да се повтарят. За разделител се използва знакът запетая.

*Пример:* Дефиницията

```
double m1[20], m2[35], proben[30];
```

е еквивалентна на дефинициите:

```
double m1[20];
double m2[35];
double proben[30];
```

Дефиницията с инициализация е един начин за свързване на променлива от тип масив с конкретна константа от множеството от стойности на този тип масив. Друг начин предоставят т.нар. **индексирани променливи**. С всяка променлива от тип масив е свързан набор от индексирани променливи. Фиг. 6.2 илюстрира техния синтаксис.

**Синтаксис на индексирани променлии**

```
<индексирана_променлива> ::=
      <променлива_от_тип_масив>[<индекс>]
```

където

<индекс> е израз от интегрален или изброен тип.

Всяка индексирана променлива е от базовия тип.

Фиг. 6.2 Синтаксис на индексираните променливи

*Примери:*

1. С променливата a, дефинирана по-горе, са свързани индексираните променливи a[0], a[1], a[2], a[3] и a[4], които са от тип int.
2. С променливата b са свързани индексираните променливи b[0], b[1],…, b[9], които са от тип double.
3. С променливата x са свързани индексираните променливи x[0], x[1],…, x[19], които са от тип enum {FALSE, TRUE}.

Дефиницията на променлива от тип масив не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет (обикновено 4B), в която записва адреса в паметта на първата индексирана променлива на масива. Останалите индексирани променливи се разполагат последователно след първата. За всяка индексирана променлива се отделя по толкова памет, колкото базовият тип изисква.

*Пример:*

ОП

a  a[0]  a[1]  …  a[4]  b  b[0]  b[1]  …  b[9] …

```
адрес     -     -           -    адрес    -      -           -
на a[0]                           на b[0]
4в       4в    4в     …    4в     4в      8в    8в    …     8в
```

За краткост, вместо “адрес на a[0]” ще записваме стрелка от a към a[0]. Стойността на отделената за индексираните променливи памет е неопределено освен ако не е зададена дефиниция с инициализация. Тогава в клетките се записват инициализиращите стойности.

*Пример:* Разпределението на паметта за променливите p и q, дефинирани в примерите по-горе, е следното:

```
ОП
p          p[0]     p[1]     p[2]     p[3]
           1.25     2.5      9.25     4.12

q          q[0]  q[1]  q[2]  q[3]  q[4]
           1       2     3    0      0
```

**Операции и вградени функции**

Не са възможни операции над масиви като цяло, но всички операции и вградени функции, които базовият тип допуска, са възможни за индексираните променливи, свързани с масива.

*Пример:* Нека

```
int a[5], b[5];
```

Недопустими са:

```
cin >> a >> b;
a = b;
```

а също a == b или a != b.

Операторът

```
cout << a;
```

е допустим и извежда адреса на a[0].

## Задачи върху тип масив

**Задача 48.** Да се напише програма, която въвежда последователно n числа, след което ги извежда в обратен ред.

Програма Zad48.cpp решава задачата.

```
// Program Zad48.cpp
#include <iostream.h>
int main()
{double x[100];
 cout << "n= ";
 int n;
 cin >> n; // въвеждане на стойност за n
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (n < 0 || n > 100)
 {cout << "Incorrect input! \n";
  return 1;
 }
 // n е цяло число от интервала [1, 100]
 // въвеждане на стойности за елементите на масива x
 for (int i = 0; i <= n-1; i++)
 {cout << "x[" << i << "]= ";
  cin >> x[i];
  if (!cin)
  {cout << "Error. Bad Input! \n";
   return 1;
  }
 } // извеждане на елементите на x в обратен ред
 for (i = n-1; i >= 0; i--)
    cout << x[i] << "\n";
 return 0;
}
```

*Изпълнение на програма Zad48.cpp*

Дефиницията double x[100]; води до отделяне на 800В ОП, които се именуват последователно с x[0], x[1], …, x[99] и са с неопределени

стойности. Освен това се отделят 4В ОП за променливата x, в които се записва адресът на индексираната променлива x[0]. Следващият програмен фрагмент въвежда стойност на n (броя на елементите на масива, които ще бъдат използвани). Операторът

```
for (int i = 0; i <= n-1; i++)
 {cout << "x[" << i << "]= ";
  cin >> x[i];
  if (!cin)
  {cout << "Error. Bad Input! \n";
   return 1;
  }
 }
```

въвежда стойности на целите променливи x[0], x[1], …, x[n-1]. Всяка въведена стойност е предшествана от подсещане. Операторът

```
for (i = n-1; i >= 0; i--)
    cout << x[i] << "\n";
```

извежда в обратен ред компонентите на масива x.

*Забележка:* Фрагментите:

```
 …                          и        …
 cout << "n= ";                      int n = 10;
 int n;                              int x[10];
 cin >> n;                           …
 int x[n];
 …
```

са недопустими, тъй като n не е константен израз. Фрагментът

```
const int n = 10;
double x[n];
```

е допустим.

## 6.3 Някои приложения на структурата от данни масив

**Търсене на елемент в редица**

Нека са дадени редица от елементи $a_0$, $a_1$, …, $a_{n-1}$, елемент x и релация r. Могат да се формулират две основни задачи, свързани с търсене на елемент в редицата, който да е в релация r с елемента x.

а) Да се намерят **всички елементи** на редицата, които са в релация r с елемента x.

б) Да се установи, **съществува ли елемент** от редицата, който е в релация r с елемента x.

Съществуват редица методи, които решават едната, другата или и двете задачи. Ще разгледаме метода на **последователното търсене**, чрез който магат да се решат и двете задачи. Методът се състои в следното: последователно се обхождат елементите на редицата и за всеки елемент се проверява дали е в релация r с елемента x. При първата задача процесът продължава до изчерпване на редицата, а при втората – до намиране на първия елемент $a_k$ (k = 0, 1, …, n-1), който е в релация r с x, или до изчерпване на редицата без да е намерен елемент с търсеното свойство.

Следващите четири задачи илюстрират този метод.

**Задача 49.** Дадени са редицата от цели числа $a_0$, $a_1$, …, $a_{n-1}$ ($n \geq 1$) и цялото число x. Да се напише програма, която намира колко пъти x се съдържа в редицата.

В случая релацията r е операцията сравнение за равенство, която се реализира чрез оператора ==. Налага се всеки елемент на редицата да бъде сравнен с x, т.е. имаме задача от първия вид. Тя описва индуктивен цикличен процес.

Програма Zad49.cpp решава задачата.

```
// Program Zad49.cpp
#include <iostream.h>
int main()
{int a[20];
 cout << "n= ";
 int n;
 cin >> n; // въвеждане на дължината на редицата
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
```

```
    if (n < 1 || n > 20)
    {cout << "Incorrect input! \n";
     return 1;
    }
    // въвеждане на редицата
    int i;
    for (i = 0; i <= n-1; i++)
    {cout << "a[" << i << "]= ";
     cin >> a[i];
     if (!cin)
     {cout << "Error. Bad input! \n";
      return 1;
     }
    }
    // въвеждане на стойност за x
    int x;
    cout << "x= ";
    cin >> x;
    if (!cin)
    {cout << "Error. Bad input! \n";
     return 1;
    }
    // намиране на броя br на срещанията на x в редицата
    int br = 0;
    for (i = 0; i <= n-1; i++)
      if (a[i] == x) br++;
    cout << "number = " << br << "\n";
    return 0;
   }
```

**Задача 50.** Дадени са редицата от цели числа $a_0$, $a_1$, …, $a_{n-1}$ ($n \geq 1$) и цялото число x. Да се напише програма, която проверява дали x се съдържа в редицата.

В този случай се изисква при първото срещане на елемент от редицата, който е равен на x, да се преустанови работата с подходящо

съобщение. Броят на сравненията на x с елементите от редицата е ограничен отгоре от n, но не е известен.

Програма Zad50.cpp решава задачата. Фрагментът, реализиращ входа, е същия като в Zad49.cpp и затова е пропуснат.

```
// Program Zad50.cpp
#include <iostream.h>
int main()
{int a[20];
 …
 i = 0;
 while (a[i] != x && i < n-1)
    i++;
 if (a[i] == x) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

Обхождането на редицата става чрез промяна на стойностите на индекса i - започват от 0 и на всяка стъпка от изпълнението на тялото на цикъла се увеличават с 1. Максималната им стойност е n-1. При излизането от цикъла ще е в сила отрицанието на условието (a[i] != x && i < n-1), т.е. (a[i] == x || i == n-1). Ако е в сила a[i] == x, тъй като сме осигурили a[i] да е елемент на редицата, отговорът “yes” е коректен. В противен случай е в сила i == n-1, т.е. сканиран е и последният елемент на редицата и за него не е вярно a[i] == x. Това е реализирано чрез отговора “no” от алтернативата на условния оператор.

Фрагментът

```
i = -1;
do
i++;
while (a[i] != x && i < n-1);
if (a[i] == x) cout << "yes \n";
else cout << "no \n";
```

реализира търсенето чрез използване на оператора do/while.

**Задача 51.** Да се напише програма, която установява, дали редицата от цели числа $a_0$, $a_1$, ..., $a_{n-1}$ е монотонно намаляваща.

а) За решаването на задачата е необходимо да се установи, дали за всяко i ($0 \le i \le n-2$) е в сила релацията a[i] >= a[i+1]. Това може да се реализира като се провери дали броят на целите числа i ($0 \le i \le n-2$), за които е в сила релацията $a[i] \ge a[i+1]$, е равен на n-1.

Програмата Zad51_1.cpp реализира този начин за проверка дали редица е монотонно намаляваща. Фрагментите, реализиращи въвеждането на n и масива a, са известни вече и затова са пропуснати.

```
// Program Zad51_1.cpp
#include <iostream.h>
int main()
{int a[100];
 // дефиниране и въвеждане на стойност на n
 …
 // въвеждане на масива a
 …
 int br = 0;
 for (i = 0; i <= n-2; i++)
    if (a[i] >= a[i+1]) br++;
 if (br == n-1) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

б) Задачата може да се сведе до търсене на i (i = 0, 1,..., n-2), така че a[I] < a[i+1], т.е. до задача за съществуване.

Програма Zad51_2.cpp реализира този начин за проверка дали редица е монотонно намаляваща. Фрагментите, реализиращи въвеждането на n и масива a отново са пропуснати.

```
// Program Zad51_2.cpp;
#include <iostream.h>
int main()
{int a[100];
```

```
 //въвеждане на размерността n и масива a
 …
 i = 0;
 while (a[i] >= a[i+1] && i < n-2) i++;
 if (a[i] >= a[i+1]) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

Решение б) е по-ефективно, тъй като при първото срещане на a[i], така че релацията a[i] < a[i+1] е в сила, изпълнението на цикъла while завършва. Решение а) реализира последователно търсене е пълно изчерпване, а решение б) – задача за съществуване на елемент в редица, който е в определена релация с друг елемент (в случая съседния му).

**Задача 52.** Да се напише програма, която установява, дали редицата от цели числа $a_0$, $a_1$, ..., $a_{n-1}$ се състои от различни елементи.

а) За решаването на задачата е необходимо да се установи, дали за всяка двойка (i, j): $0 \le i \le n-2$ и $i+1 \le j \le n-1$ е в сила релацията a[i] != a[j]. Това може да се постигне като се провери дали броят на двойките (i, j): $0 \le i \le n-2$ и $i+1 \le j \le n-1$, за които е в сила релацията a[i] != a[j], е равен на n*(n-1)/2.

Програма Zad52_1.cpp реализира тази идея за решение на задачата – търсене с пълно изчерпване. Фрагментите, реализиращи въвеждането на n и масива a, отново са пропуснати.

```
// Program Zad52_1.cpp
#include <iostream.h>
int main()
{int a[100];
 //въвеждане на размерността n и масива a
 …
 int br = 0;
 for (i = 0; i <= n-2; i++)
    for (int j = i+1; j <= n-1; j++)
```

```
        if (a[i] != a[j]) br++;
    if (br == n*(n-1)/2) cout << "yes \n";
    else cout << "no \n";
    return 0;
   }
```

б) Задачата може да се сведе до проверка за съществуване на двойка индекси (i, j): $0 \le i \le n-2$ и $i+1 \le j \le n-1$, за които не е в сила релацията a[i] != a[j]. Програма Zad52_2.cpp реализира тази идея.

```
   // Program Zad52_2.cpp
   #include <iostream.h>
   int main()
   {int a[100];
    // въвеждане стойности на размерността n и масива a
    …
    i = -1;
    int j;
    do
    {i++;
     j = i+1;
     while (a[i] != a[j] && j < n-1) j++;
    } while (a[i] != a[j] && i < n-2);
    if (a[i] != a[j]) cout << "yes \n";
    else cout << "no \n";
    return 0;
   }
```

Решение б) е по-ефективно, тъй като при първото срещане на a[i] и a[j], така че релацията a[i] == a[j] е в сила, изпълнението на операторите за цикъл завършва. То реализира задача за съществуване на метода за търсене.

**Сортиране на редица**

Да се сортира редицата $a_0$, $a_1$, ..., $a_{n-1}$ означава така да се пренаредят елементите ѝ, че за новата редица да е в сила $a_0 \le a_1 \le \ldots \le a_{n-1}$ или $a_0 \ge a_1 \ge \ldots \ge a_{n-1}$. В първия случай се казва, че

редицата е сортирана във **възходящ**, а във втория случай, че е сортирана в **низходящ ред**.

Съществуват много методи за сортиране на редици от елементи. В тази глава ще разгледаме **метода на пряката селекция** и чрез него ще реализираме възходяща сортировка на редица.

*Метод на пряката селекция*

Разглежда се редицата $a_0$, $a_1$, …, $a_{n-1}$ и се извършват следните действия:

- Намира се k, така че $a_k = \min\{a_0, a_1, \ldots, a_{n-1}\}$.
- Разменят се стойностите на $a_k$ и $a_0$.

Така на първо място в редицата се установява най-малкият й елемент.

Разглежда се редицата $a_1$, $a_2$, …, $a_{n-1}$ и се извършват действията:

- Намира се k, така че $a_k = \min\{a_1, a_2, \ldots, a_{n-1}\}$.
- Разменят се стойностите на $a_k$ и $a_1$.

Така на второ място в редицата се установява следващият по големина елемент на редицата и т.н.

Разглежда се редицата $a_{n-2}$, $a_{n-1}$ и се извършват действията:

- Намира се k, така че $a_k = \min\{a_{n-2}, a_{n-1}\}$.
- Разменят се стойностите на $a_k$ и $a_{n-2}$.

Получената редица е сортирана във възходящ ред.

**Задача 53.** Да се сортира във възходящ ред по метода на пряката селекция числовата редица $a_0$, $a_1$, …, $a_{n-1}$ ($n \geq 1$).

Програма Zad53.cpp решава задачата. Фрагментът за въвеждане стойности на размерността n и масива a отново е пропуснат.

```
// Program Zad53.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[100];
 ...
 int i;
```

```
 for (i = 0; i <= n-2; i++)
 {int min = a[i];
  int k = i;
  for (int j = i+1; j <= n-1; j++)
    if (a[j] < min)
    {min = a[j];
     k = j;
    }
  int x = a[i]; a[i] = a[k]; a[k] = x;
 }
 for (i = 0; i <= n-1; i++)
   cout << setw(10) << a[i];
 cout << '\n';
 return 0;
}
```

**Сливане на редици**

Сливането е вид сортиране. Нека са дадени сортираните във възходящ ред редици:

$a_0, a_1, \ldots, a_{n-1}$

$b_0, b_1, \ldots, b_{m-1}$

Да се слеят редиците означава да се конструира нова, сортирана във възходящ ред редица, съставена от елементите на дадените редици. Осъществява се по следния начин:

- Поставят се “указатели” към първите елементи на редиците $\{a_i\}$ и $\{b_j\}$.
- Докато има елементи и в двете редици, се сравняват елементите, сочени от “указателите”. По-малкият елемент се записва в новата редица, след което се прескача.
- След изчерпване на елементите на едната от дадените редици, елементите на другата от “указателя” (включително) се прехвърлят в новата редица.

**Задача 54.** Дадени са сортираните във възходящ ред редици:

$a_0, a_1, \ldots, a_{n-1}$

$b_0, b_1, \ldots, b_{m-1}$

($n \geq 1$, $m \geq 1$). Да се напише програма, която слива двете редици в редицата

$c_0, c_1, \ldots, c_{k-1}$.

Програма Zad54.cpp решава задачата.

```
// Program Zad54.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[20];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 20)
 {cout << "Incorrect input! \n";
  return 1;
 }
 int i;
 for (i = 0; i <= n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
 int b[10];
 cout << "m= ";
 int m;
 cin >> m;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (m < 1 || m > 10)
 {cout << "Incorrect input! \n";
```

```
 return 1;
}
for (i = 0; i <= m-1; i++)
{cout << "b[" << i << "]= ";
 cin >> b[i];
}
int p1 = 0, p2 = 0;
int c[30];
int p3 = -1;
while (p1 <= n-1 && p2 <= m-1)
if (a[p1] <= b[p2])
{p3++;
 c[p3] = a[p1];
 p1++;
}
else
{p3++;
c[p3] = b[p2];
p2++;
}
if (p1 > n-1)
   for (i = p2; i <= m-1; i++)
    {p3++;
     c[p3] = b[i];
    }
else
   for (i = p1; i <= n-1; i++)
    {p3++;
     c[p3] = a[i];
    } // извеждане на редицата
 for (I = 0; i <= p3; i++)
   cout << setw(10) << c[i];
 cout << '\n';
 return 0;
}
```

Разглежданите досега масиви се наричат **едномерни**. Те реализират крайни редици от елементи от скаларен тип. Възможно е обаче типът на елементите да е масив. В този случай се говори за многомерни масиви.

## 6.4 Многомерни масиви

Масив, базовият тип на който е едномерен масив, се нарича **двумерен**. Масив, базовият тип на който е двумерен масив, се нарича **тримерен** и т.н. На практика се използват масиви с размерност най-много 3.

**Дефиниране на многомерни масиви**

Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален, $size_1$, $size_2$, …, $size_n$ (n>1 е дадено цяло число) са константни изрази от интегрален или изброен тип с положителни стойности. T[$size_1$][$size_2$] … [$size_n$] е тип n-мерен масив от тип T. T се нарича базов тип за типа масив.

*Примери:*

int [5][3]  дефинира двумерен масив от тип int;

double [4][5][3]  дефинира тримерен масив от тип double;

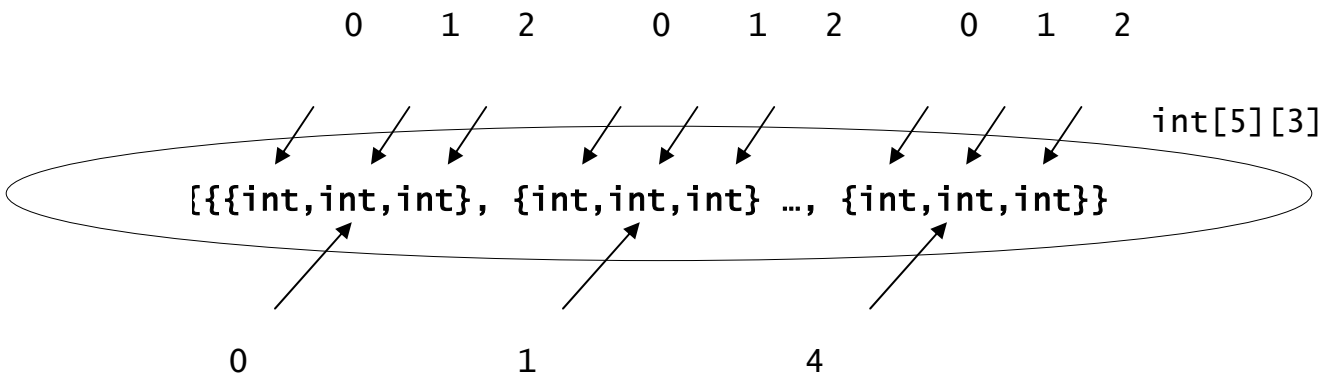
**Множество от стойности**

Множеството от стойности на типа T[$size_1$][$size_2$] … [$size_n$] се състои от всички редици от по $size_1$ елемента, които са произволни константи от тип T[$size_2$] … [$size_n$]. Достъпът до елементите на редиците е пряк и се осъществява с помощта на индекс, като достъпът до първия елемент се осъществява с индекс със стойност 0, до последния – с индекс със стойност $size_1$-1, а до всеки от останалите елементи – с индекс със стойност с 1 по-голяма от тази на индекса на предишния елемент. Елементите от множеството от стойности на даден тип многомерен масив са **константите** на този тип масив.

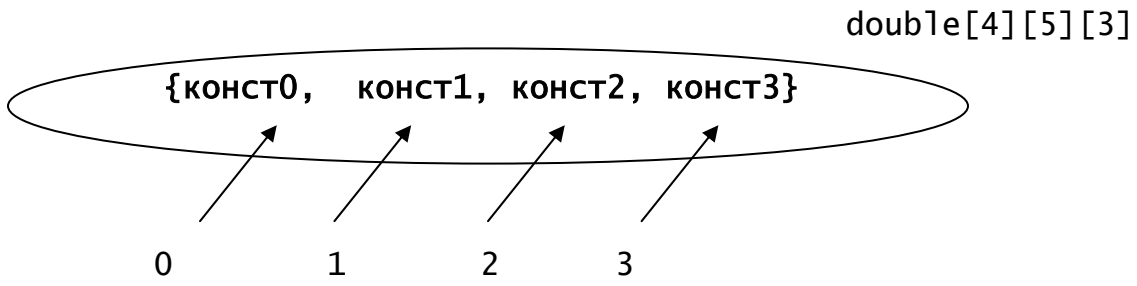
*Примери:*

1. Множеството от стойности на типа int[5][3] се състои от всички редици от по 5 елемента, които са едномерни масиви от тип int[3].

Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2, 3 и 4.




2. Множеството от стойности на типа double[4][5][3] се състои от всички редици от по 4 константи от тип double[5][3]. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2 и 3.




където с конст$_i$ (i = 0, 1, 2, 3) е означена произволна константа от тип double[5][3].

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип масив, се нарича променлива от дадения тип масив или само масив. Фиг. 6.3 дава обобщение на синтаксиса на дефиницията на променлива от тип масив.

**Синтаксис на многомерен масив**

<дефиниция_на_променлива_от_тип_многомерен_масив> ::=
  T <променлива>[size$_1$][size$_2$]…[size$_n$]
    [= {<редица_от_константи_от_тип T1>}]$_{опц}$
  {,<променлива>[size$_1$][size$_2$]…[size$_n$]
    [= {<редица_от_константи_от_тип T1>}]$_{опц}$}$_{опц}$;

```
        {,<променлива>[size1][size2]…[sizen]
            [= {<редица_от_константи_от_тип T>}]опц}опц;
```

където

- T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален;
- T1 е име на типа T[$size_2$]…[$size_n$];
- $size_1$, $size_2$,…, $size_n$ са константни изрази от интегрален или изброен тип със *положителни* стойности;
- <променлива> е идентификатор;
- <редица_от_константи_от_тип T1> се дефинира по следния начин:
  <редица_от_константи_от_тип T1> ::= <константа_от_тип T1>|
  <константа_от_тип T1>, <редица_от_константи_от_тип T1>
- <редица_от_константи_от_тип T> се определя по аналогичен начин.

Фиг. 6.3 Синтаксис на многомерен масив

*Примери:*

```
int x[10][20];
double y[20][10][5];
int z[3][2] = {{1, 3},
               {5, 7},
               {2, 9}};
int t[2][3][2] = {{{1, 3}, {5, 7}, {6, 9}},
                  {{7, 8}, {1, 8}, {-1, -4}}};
```

Ще отбележим, че фрагментите:

<променлива>[$size_1$][$size_2$] … [$size_n$],

<променлива>[$size_1$][$size_2$]…[$size_n$]={<редица_от_константи_от_тип T1>}

<променлива>[$size_1$][$size_2$]…[$size_n$]={<редица_от_константи_от_тип T>}

от дефиницията от Фиг. 6.3, могат да се повтарят. За разделител се използва символът запетая.

*Примери:*

```
int a[3][4], b[2][3][2] = {{{1, 2}, {3, 4}, {5, 6}},
                           {{7, 8}, {9, 0}, {1, 2}}};
double c[2][3] = {1, 2, 3, 4, 5, 6}, d[3][4][5][6];
```

При дефиницията с инициализация, от Фиг. 6.3, е възможно $size_1$ да се пропусне. Тогава за стойност на $size_1$ се подразбира броят на

редиците от константи на най-външно ниво, изброени при инициализацията.

*Пример:* Дефиницията

```
int s[][2][3] = {{{1,2,3}, {4, 5, 6}},
                 {{7, 8, 9}, {10, 11, 12}},
                 {{13, 14, 15}, {16, 17, 18}}};
```

е еквивалентна на

```
int s[3][2][3] = {{{1,2,3}, {4, 5, 6}},
                  {{7, 8, 9}, {10, 11, 12}},
                  {{13, 14, 15}, {16, 17, 18}}};
```

Ако изброените константни изрази в инициализацията на ниво i са по-малко от $size_i$, останалите се инициализират с нулеви стойности.

*Примери:* Дефиницията

```
int fi[5][6] = {{1, 2}, {5}, {3, 4, 5},
                {2, 3, 4, 5}, {2, 0, 4}};
```

е еквивалентна на

```
int fi[5][6] = {{1, 2, 0, 0, 0, 0},
                {5, 0, 0, 0, 0, 0},
                {3, 4, 5, 0, 0, 0},
                {2, 3, 4, 5, 0, 0},
                {2, 0, 4, 0, 0, 0}};
```

Вложените фигурни скоби не са задължителни. Следното инициализиране

```
int ma[4][3] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

е еквивалентно на

```
int ma[4][3] = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}};
```

но е по-неясно.

Следващата дефиниция

```
int ma[4][3] = {{0}, {1}, {2}, {3}};
```

е еквивалентна на

```
int ma[4][3] = {{0, 0, 0 }, {1, 0, 0}, {2, 0, 0}, {3, 0, 0}};
```

и е различна от

```
int ma[4][3] = {0, 1, 2, 3};
```

която пък е еквивалентна на

```
int ma[4][3] = {{0, 1, 2}, {3, 0, 0}, {0, 0, 0}, {0, 0, 0}};
```

Инициализацията е един начин за свързване на променлива от тип масив с конкретна константа от множеството от стойности на този тип масив. Друг начин дават индексираните променливи. С всяка променлива от тип масив е свързан набор от индексирани променливи. Фиг. 6.4 обобщава техния синтаксис.

**Синтаксис на индексирана променлива от тип многомерен масив**

<индексирана_променлива> ::=

<променлива_от_тип_масив>[<индекс$_1$>][<индекс$_2$>] … [<индекс$_n$>]

където

<индекс$_i$> е *израз* от интегрален или изброен тип.

Всяка индексирана променлива е от базовия тип.

Фиг. 6.4 Синтаксис на индексирана променлива от тип многомерен масив

*Примери:*

1. С променливата x, дефинирана по-горе, са свързани индексираните променливи

```
x[0][0],    x[0][1],    …, x[0][19],
x[1][0],    x[1][1],    …, x[1][19],
                        …
x[9][0],    x[9][1],    …, x[9][19],
```

които са от тип int.

2. С променливата y са свързани следните реални индексирани променливи:

```
y[i][0][0], y[i][0][1], …, y[i][0][4],
y[i][1][0], y[i][1][1], …, y[i][1][4],
                        …
y[i][9][0], y[i][9][1], …, y[i][9][4],
```

за i = 0, 1, …, 19.

Дефиницията на променлива от тип многомерен масив не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет за разполагане на индексираните променливи, свързани с нея. Индексираните променливи се разполагат последователно по по-бързото нарастване на по-далечните си индекси. За всяка индексирана променлива се отделя толкова памет, колкото

базовият тип изисква. Чрез пример ще илюстрираме по-подробно представянето.

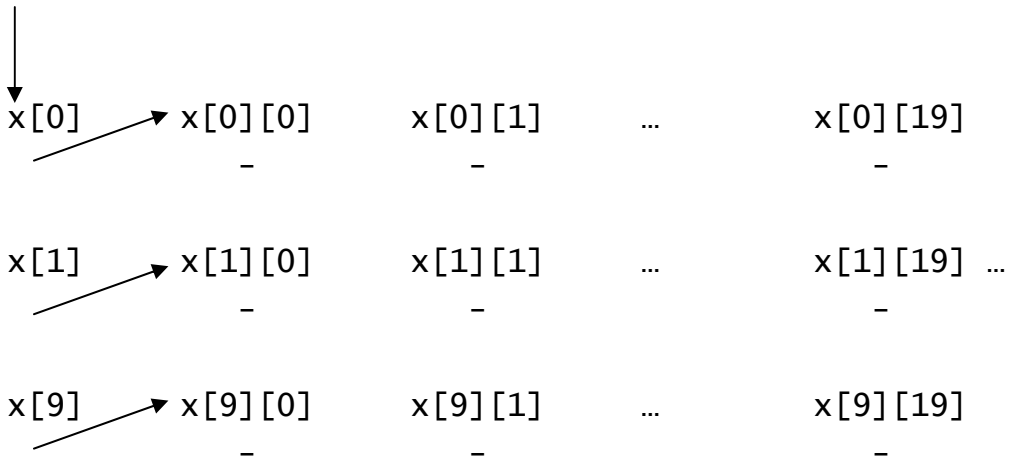
След дефиницията

int x[10][20];

за променливата x се отделят 4B, в които се записва адресът на първата компонента, в случая x[0], на едномерен масив, елементите на който са адреси. Стойността на индексираната променлива x[i] е адреса на индексираната променлива x[i][0] (i = 0, 1, ..., 9), т.е.

ОП

x




Операторът

cout << x[0] << x[1] << ... << x[9];

ще изведе адресите на x[0][0], x[1][0], ... и x[9][0].

Индексираните променливи x[i][j] (i=0, 1, ..., 9; j = 0, 1, ..., 19) са променливи от тип int и за всяка от тях са отделени 4B ОП, които са с неопределени стойности, тъй като x не е дефинирана с инициализация.

**Забележка:** Двумерните масиви разполагат в ОП индексираните си променливи по по-бързото нарастване на втория индекс. Това физическо представяне се нарича **представяне по редове**. Тези масиви могат да бъдат използвани за реализация и работа с матрици и др. правоъгълни таблици.

**Важно допълнение:** При работа с масиви трябва да се има предвид, че повечето реализации не проверяват дали стойностите на индексите са в рамките на границите, зададени при техните дефиниции. Тази особеност крие опасност от допускане на труднооткриваеми грешки.

**Често допускана грешка:** В Паскал, Ада и др. процедурни езици, индексите на индексираните променливи се ограждат само в една двойка квадратни скоби и се отделят със запетаи. По навик, при програмиране на C++, често се използва същото означение. Това е неправилно, но за съжаление не винаги е съпроводено със съобщение за грешка, тъй като в езика C++ съществуват т.нар. comma-изрази. Използвахме ги вече в заглавните части на оператора за цикъл for. Comma-изразите са изрази, отделени със запетаи. Стойността на най-десния израз е стойността на comma-израза. Операторът за последователно изпълнение запетая е лявоасоциативен. Така 1+3, 8, 21-15 е comma-израз със стойност 6, а [1, 2] е comma-израз със стойност [2]. В C++ ma[1,2] означава адреса на индексираната променлива ma[2][0] (индексът [0] се добавя автоматично).

## Задачи върху многомерни масиви

**Задача 55.** Да се напише програма, която въвежда елементите на правоъгълна матрица a[nxm] от цели числа и намира и извежда матрицата, получена от дадената като всеки от нейните елементи е увеличен с 1.

Програма Zad55.cpp решава задачата.

```
// Program Zad55.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[10][20];
 // въвеждане на броя на редовете на матрицата
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 10)
```

```
 {cout << "Incorrect input! \n";
  return 1;
 }
// въвеждане на броя на стълбовете на матрицата
 cout << "m= ";
 int m;
 cin >> m;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (m < 1 || m > 20)
 {cout << "Incorrect input! \n";
  return 1;
 }
 // въвеждане на матрицата по редове
 int i, j;
 for (i = 0; i <= n-1; i++)
    for (j = 0; j <= m-1; j++)
    {cout << "a[" << i << ", " << j << "]= ";
     cin >> a[i][j];
     if (!cin)
     {cout << "Error. Bad Input! \n";
      return 1;
     }
    }
 // конструиране на нова матрица b
 int b[10][20];
 for (i = 0; i <= n-1; i++)
    for (j = 0; j <= m-1; j++)
       b[i][j] = a[i][j] + 1;
 // извеждане на матрицата b по редове
 for (i = 0; i <= n-1; i++)
 {for (j = 0; j <= m-1; j++)
   cout << setw(6) << b[i][j];
   cout << '\n';
```

```
 }
 return 0;
}
```

**Забележка:** За реализиране на операциите въвеждане, извеждане и конструиране се извърши обхождане на елементите на двумерен масив по редове.

**Задача 56.** Да се напише програма, която намира и извежда сумата от елементите на всеки стълб на квадратната матрица a[nxn].

Програма Zad56.cpp решава задачата.

```
// Program Zad56.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[10][10];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 10)
 {cout << "Incorrect input! \n";
  return 1;
 }
 int i, j;
 for (i = 0; i <= n-1; i++)
    for (j = 0; j <= n-1; j++)
    {cout << "a[" << i << ", " << j << "]= ";
     cin >> a[i][j];
     if (!cin)
     {cout << "Error. Bad Input! \n";
      return 1;
     }
```

```
      }
    for (j = 0; j <= n-1; j++)
    {int s = 0;
     for (i = 0; i <= n-1; i++)
       s += a[i][j];
     cout << setw(10) << j << setw(10) << s << "\n";
    }
    return 0;
   }
```

Реализирано е обхождане на масива по стълбове (първият индекс се изменя по-бързо).

**Задача 57.** Да се напише програмен фрагмент, който намира номерата на редовете на целочислената квадратна матрица a[nxn], в които има елемент, равен на цялото число x.

```
 …
 for (i = 0; i <= n-1; i++)
 {j = -1;
  do
  j++;
  while (a[i][j] != x && j < n-1);
  if (a[i][j] == x) cout << setw(5) << i << '\n';
  }
 …
```

Фрагментът реализира последователно обхождане на *всички* редове на матрицата и за всеки ред проверява дали *съществува* елемент, равен на дадения елемент x.

**Задача 58.** Да се напише програмен фрагмент, който обхожда кватратната матрица a[nxn] по диагонали, започвайки от елемента $a_{00}$, както е показано по-долу:

$a_{00}$ $a_{01}$ $a_{02}$ … $a_{0,n-1}$

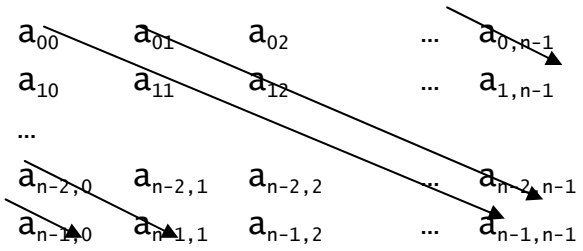
$a_{10}$ $a_{11}$ $a_{12}$ … $a_{1,n-1}$

…

$a_{n-1,0}$ $a_{n-1,1}$ $a_{n-1,2}$ … $a_{n-1,n-1}$

```
…
int k;
for (k = 0; k <= n-1; k++)
{for (i = k; i >= 0; i--)
 cout << "(" << i << ", "<< k-i << ") ";
 cout << '\n';
}
for (k = n; k <= 2*n-2; k++)
 {for (i = n-1; i >= k-n+1; i--)
  cout << "(" << i << ", "<< k-i << ") ";
  cout << '\n';
 }
…
```

**Задача 59.** Да се напише програмен фрагмент, който обхожда кватратната матрица a[nxn] по диагонали, започвайки от елемента $a_{n-1,0}$, както е показано по-долу:




```
 …
int k;
for (k = n-1; k >= 0; k--)
{for (i = k; i <= n-1; i++)
 cout << "(" << i << ", "<< i-k << ") ";
 cout << '\n';
}
for (k = -1; k >= 1-n; k--)
{for (i = 0; i <= n+k-1; i++)
 cout << "(" << i << ", "<< i-k << ") ";
 cout << '\n';
}
```

…

**Задача 60.** Да се напише програма, която:

а) въвежда по редове елементите на квадратната реална матрица A с размерност nxn;

б) от матрицата A конструира редицата B: $b_0$, $b_2$,..., $b_{m-1}$, където m = $n^2$, при което първите n елемента на B съвпадат с елементите на първия стълб на A, вторите n елемента на B съвпадат с елементите на втория стълб на A и т.н., последните n елемента на B съвпадат с елементите на последния стълб на A;

в) сортира във възходящ ред елементите на редицата B;

г) образува нова квадратна матрица A с размерност nxn, като елементите от първия ред на A съвпадат с първите n елемента на B, елементите от втория ред на A съвпадат с вторите n елемента на B и т.н. елементите от n - тия ред на A съвпадат с последните n елемента на B;

д) извежда по редове новата матрица A.

Програма Zad60.cpp решава задачата.

```
// Program Zad60.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[10][10];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 10)
 {cout << "Incorrect input! \n";
  return 1;
 }
 // въвеждане на масива a
```

```
int i, j;
for (i = 0; i <= n-1; i++)
    for (j = 0; j <= n-1; j++)
   {cout << "a[" << i<< "][" << j  << "]= ";
    cin >> a[i][j];
   }
// извеждане на елементите на a по редове
for (i = 0; i <= n-1; i++)
{for (j = 0; j <= n-1; j++)
  cout << setw(5) << a[i][j];
 cout << "\n";
}
// развиване на матрицата a по стълбове
int b[100];
int m = -1;
for (j = 0; j <= n-1; j++)
   for (i = 0; i <= n-1; i++)
   {m++;
    b[m] = a[i][j];
   }
   m++; // m е броя на елементите на редицата b
// извеждане на редицата b
for (i = 0; i <= m-1; i++)
   cout << setw(5) << b[i];
cout << '\n';
// сортиране на b по метода на пряката селекция
for (i = 0; i <= m-2; i++)
{int k = i;
 int min = b[i];
 for (j = i+1; j <= m-1; j++)
   if (b[j] < min)
   {min = b[j];
    k = j;
   }
 int x = b[i]; b[i] = b[k]; b[k] = x;
}
```

```
    // извеждане на сортираната b
    for (i = 0; i <= m-1; i++)
       cout << setw(5) << b[i];
    cout << '\n';
    // конструиране на новата матрица a
    m = -1;
    for (i = 0; i <= n-1; i++)
       for (j = 0; j <= n-1; j++)
       {m++;
       a[i][j] = b[m];
       }
    // извеждане на матрицата a
    for (i = 0; i <= n-1; i++)
      {for (j = 0; j <= n-1; j++)
         cout << setw(10) << a[i][j];
       cout << '\n';
      }
    return 0;
   }
```

## 6.5 Символни низове

### 6.5.1. Структура от данни низ

**Логическо описание**

Крайна, евентуално празна редица от символи, заградени в кавички, се нарича **символен низ**, **знаков низ** или само **низ**.

Броят на символите в редицата се нарича **дължина** на низа. Низ с дължина 0 се нарича **празен**.

*Примери:* “xyz” е символен низ с дължина 3,
“This is a string.” е символен низ с дължина 17, а
“” е празния низ.

Низ, който се съдържа в даден низ се нарича негов **подниз**.

*Пример:* Низът “ is s “ е подниз на низа “This is a string.”, а низът “ is a sing” не е негов подниз.

**Конкатенация на два низа** е низ, получен като в края на първия низ се запише вторият. Нарича се още **слепване** на низове.

*Пример:* Конкатенацията на низовете “a+b” и “=b+a” е низът “a+b=b+a”, а конкатенацията на “=b+a” с “a+b” е низът “=b+aa+b”. Забелязваме, че редът на аргументите е от значение.

**Два символни низа се сравняват** по следния начин: Сравнява се всеки символ от първия низ със символа от съответната позиция на втория низ. Сравнението продължава до намиране на два различни символа или до края на поне един от символните низове. Ако кодът на символ от първия низ е по-малък от кода на съответния символ от втория низ, или първият низ е изчерпен, приема се, че първият низ е по-малък от втория. Ако пък е по-голям или вторият низ е изчерпен – приема се, че първият низ е по-голям от втория. Ако в процеса на сравнение и двата низа едновременно са изчерпени, те са равни. Това сравнение се нарича **лексикографско**.

*Примери:* “abbc” е равен на “abbc”
“abbc” е по-малък от “abbcaaa”
“abbc” е по-голям от “aa”
“abbcc” е по-голям от “abbc”.

**Физическо представяне**

Низовете се представят последователно.

### 6.5.2 Символни низове в езика C++

Съществуват два начина за разглеждане на низовете в езика C++:

- като едномерни масиви от символи;
- като указатели към тип char.

За щастие, те са семантично еквивалентни.

В тази част ще разгледаме символните низове като масиви от символи.

Дефиницията

```
char str1[100];
```

определя променливата str1 като масив от 100 символа, а

```
char str2[5] = {‘a’, ‘b’, ‘c’};
```

дефинира масива от символи str2 и го инициализира. Тъй като при инициализацията са указани по-малко от 5 символа, останалите се допълват с нулевия символ, който се означава със символа \0, а понякога и само с 0. Така последната дефиниция е еквивалентна на дефинициите:

```
char str2[5] = {‘a’, ‘b’, ‘c’, ‘\0’, ‘\0’};
char str2[5] = {‘a’, ‘b’, ‘c’, 0, 0};
```

Всички действия за работа с едномерни масиви, които описахме в т. 2 и 4, са валидни и за масиви от символи с изключение на извеждането. Операторът

```
cout << str2;
```

няма да изведе адреса на str2 (както е при масивите от друг тип), а текста

```
abc
```

Има обаче една особеност. Ако инициализацията на променливата str2 е пълна (съдържа точно 5 симвала) и не завършва със символа \0, т.е. има вида

```
char str2[5] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’};
```

операторът

```
cout << str2;
```

извежда текста

```
abcde<неопределено>
```

Имайки пред вид това, бихме могли да напишем подходящи програмни фрагменти, които въвеждат, извеждат, копират, сравняват, извличат части, конкатенират низове. Тъй като операциите се извършват над индексираните променливи, налага се да се поддържа целочислена променлива, съдържаща дължината на низа.

В езика са дадени средства, реализиращи низа като скаларна структура. За целта низът се разглежда като редица от символи, завършваща с нулевия символ \0, наречен още **знак за край на низ**. Тази организация има предимството, че не е необходимо с всеки низ да се пази в променлива дължината му, тъй като знакът за край на низ позволява да се определи краят му.

*Примери:* Дефинициите

```
char s1[5] = {‘a, ‘b’, ‘b’, ‘a’, ‘\0’};
char s2[10] = {‘x’, ‘y’, ‘z’, ‘1’, ‘2’, ‘+’, ‘\0’};
```

свързват променливите s1 и s2 от тип масив от символи с низовете “abba” и “xyz12+” съответно. Знакът за край на низ ‘\0’ не се включва явно в низа.

Този начин за инициализация не е много удобен. Следните дефиниции са еквивалентни на горните.

char s1[5] = “abba”;

char s2[10] = “xyz12+”;

Забелязваме, че ако низ, съдържащ n символа, трябва да се свърже с масив от символи, минималната дължина на масива трябва да бъде n+1, за да се поберат n-те символа и символът \0.

### 6.5.3 Тип символен низ

**Дефиниране**

Типът char[size], където size е константен израз от интегрален или изброен тип, **може да бъде използван за задаване на тип низ с максимална дължина size-1**.

*Пример:*

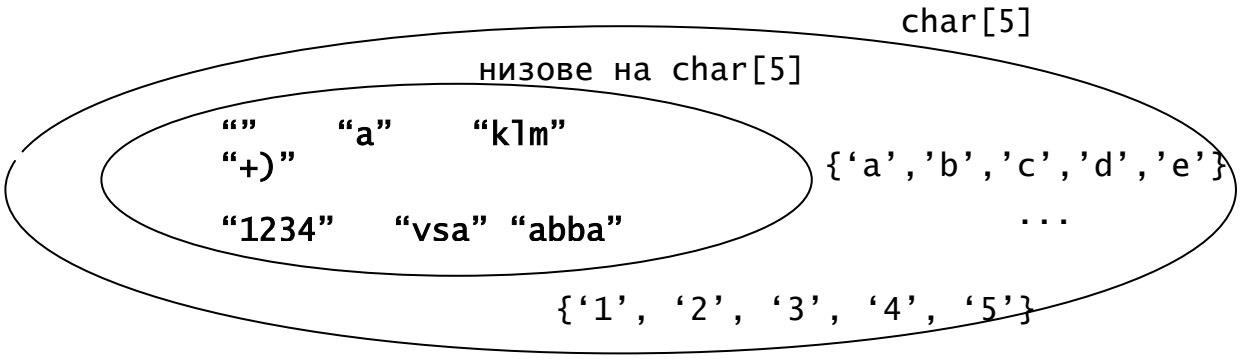
char[5] може да се използва за задаване на тип низ с максимална дължина 4.

**Множество от стойности**

Множеството от стойности на типа низ, зададен чрез char[size], се състои от всички низове с дължина 0, 1, 2, ..., size-1. То е подмножество на множеството от стойности на типа char[size].

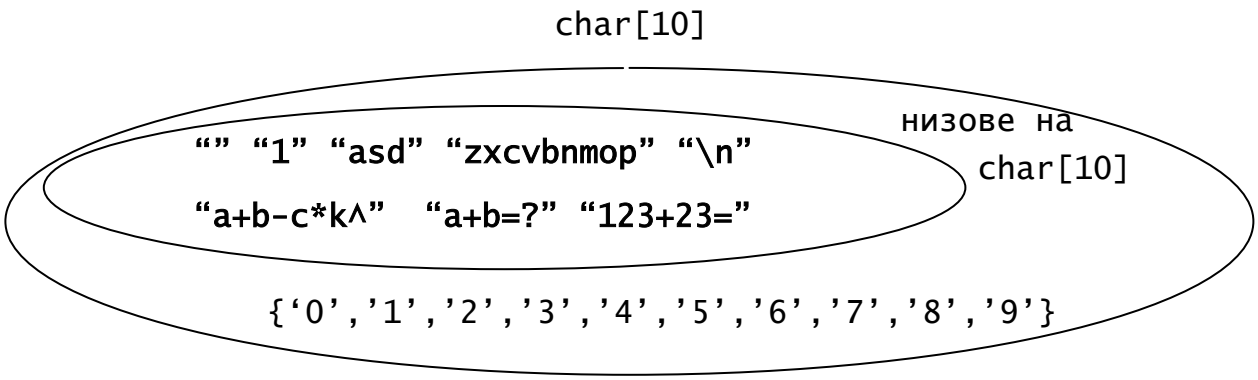
*Примери:*

1. Множеството от стойности на типа низ, зададен чрез char[5] се състои от всички низове с дължина 0, 1, 2, 3 и 4.



*Забележка:* Редицата {‘\0’, ‘\0’, ‘\0’, ‘\0’, ‘\0’} представя празния низ “”,{c, ‘\0’, ‘\0’, ‘\0’, ‘\0’}, където c е произволен символ представя низ с дължина 1, {c, c, ‘\0’, ‘\0’, ‘\0’} е низ с дължина 2 и т.н.

2. Множеството от стойности на типа char[10] се състои от всички низове с дължина 0, 1, 2, ..., 9.




Елементите от множеството от стойности на даден тип низ са неговите **константи**. Например, “a+b=c-a*e”, “1+3”, “” са константи от тип char[10].

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип низ, се нарича променлива от този тип низ. Понякога ще я наричаме само низ.

Фиг. 6.5 определя дефиницията на променлива от тип низ.

**Дефиниция на променливи от тип символен низ**

<дефиниция_на_променлива_от_тип_низ> ::=

char <променлива>[size] [= “<редица_от_символи>”|

= {<редица_от_константни_изрази>}]$_{опц}$

{,<променлива>[size] [= “<редица_от_символи>”|

={<редица_от_константни_изрази>}]$_{опц}$}$_{опц}$;

където

- <променлива> е идентификатор;
- size е константен израз от интегрален или изброен тип със *положителна* стойност;
- редица от константни изрази се дефинира по следния начин:

```
   <редица_от_константни_изрази> ::= <константен_израз>|
         <константен_израз>, <редица_от_константни_изрази>
като константните изрази в случая са от тип char;
   - редица от символи се определя по следния начин:
   <редица_от_символи> ::= <празно>| <символ> |
         <символ><редица_от_символи>
като максималната й дължина е size-1.
```

Фиг. 6.5 Дефиниция на променливи от тип символен низ

Ще отбележим,че фрагментите:

<променлива>[size]

<променлива>[size] = “<редица_от_символи>” и

<променлива>[size] = {<редица_от_константни_изрази>}

могат да се повтарят. За разделител се използва знакът запетая.

*Примери:*

char s1[5], t[12] = “12345+34”;

char s2[10] = “x+y”, s4, s5 = {‘3’, ‘5’, ‘\0’};

char s3[8] = {‘1’, ‘2’, ‘3’, ‘\0’};

При дефиниция на низ с инициализация е възможно size да се пропусне. Тогава инициализацията трябва да съдържа символа ‘\0’ и за стойност на size се подразбира броят на константните изрази, изброени при инициализацията, включително ‘\0’. Ако size е указано, изброените константни изрази в инициализацията може да са по-малко от size. Тогава останалите се инициализират с ‘\0’.

*Примери:*

Дефиницията

char q[5] = {‘a’, ‘b’};

е еквивалентна на

char q[5] = {‘a’, ‘b’, ‘\0’, ‘\0’, ‘\0’};

а също и на

char q[5] = “ab”;

а

char r[] = {‘a’, ‘b’, ‘\0’}; или

```
char r[] = “ab”;
```

са еквивалентни на

```
char r[3] = {‘a’, ‘b’, ‘\0’}; или
char r[3] = “ab”;
```

**Забележка:** Не се допускат конструкции от вида:

```
char q[5];
q = {‘a’, ‘v’, ‘s’}; или
char r[5];
r = “avs”;
```

т.е. **на променлива от тип низ не може да бъде присвоявана константа от тип низ.**

Недопустими са също дефиниции от вида:

```
char q[4] = {‘a’, ‘s’, ‘d’, ‘f’, ‘g’, ‘h’}; или
char q[];
```

Инициализацията е един начин за свързване на променлива от тип низ с конкретна константа от множеството от стойности на този тип низ. Друг начин предоставят индексираните променливи.

*Примери:*

```
q[0] = ‘a’; q[1] = ‘s’; q[2] = ‘d’;
```

Дефиницията на променлива от тип низ не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет (обикновено 4В), в която записва адреса на първата индексирана променлива, свързана с променливата от тип низ. Останалите индексирани променливи се разполагат последователно след първата. За всяка индексирана променлива се отделя по 1В ОП. Стойността на отделената за индексираните променливи памет е неопределена освен ако не е зададена дефиниция с инициализация. Тогава в клетките се записват инициализиращите стойности, допълнени със знака за край на низ.

*Пример:* След дефиницията

```
char s[4];
char w[10] = “abba”;
```

разпределението на паметта има вида:

```
ОП
s        s[0]  s[1]  s[2]  s[3]
         -     -     -     -
```

```
w     w[0]  w[1] w[2]   w[3]  w[4] w[5]   ...   w[9] …
       97    98   98     97    0    0           0
```

**Операции и вградени функции**

**Въвеждане на стойност**

Реализира се по стандартния начин - чрез оператора cin.

*Пример:*

```
char s[5], t[3];
cin >> s >> t;
```

Настъпва пауза в очакване да се въведат два низа с дължина **не по-голяма** от 4 в първия и не по-голяма от 2 във втория случай. Водещите интервали, табулации и знакът за преминаване на нов ред се пренебрегват. За разделител на низовете се използват интервалът, табулациите и знакът за преминаване на нов ред. Знакът за край на низ автоматично се добавя в края на всяка от въведените знакови комбинации.

**Забележка**: При въвеждане на низовете не се извършва проверка за достигане на указаната горна граница. Това може да доведе до трудноотриваеми грешки.

Друг начин за въвеждане на низове дава функцията getline. В тази глава ще я разгледаме непълно и малко неточно. Фиг. 6.6 описва нейните синтаксис и семантика.

**Функция getline**

*Синтаксис*

cin.getline(<var_str>, <size>, [<char>]$_{опц}$)

където

- <var_str> е променлива от тип низ;
- <size> е цял израз;
- <char> е произволен символ.

Ако <char> е пропуснато, подразбира се символът \n.

*Семантика*

Въвежда от буфера на клавиатурата редица от символи с максимална дължина <size>-1. Въвеждането продължава до срещане на символа, зададен в <char> или до въвеждане на <size>-1 символа (ако междувременно не е достигнат символът от <char>).

Фиг. 6.6 Функция getline

*Примери:*

1. char s1[100];
   cin.getline(s1, 10);

Настъпва пауза. Очаква се въвеждането на низ, след което да се натисне клавиша ENTER. Ако дължината на въведения низ е по-голяма от 9 преди да е натиснат клавишът ENTER, вземат се първите 9 символа, свързват се с променливата s1 и изпълнението завършва. В противен случай, в s1 се записва редицата от символи без символа \n.

2. char s2[200];
   cin.getline(s2, 200, '.');

Настъпва пауза. Очаква се въвеждането на низ, което да завърши с натискане на клавиша ENTER. Ако дължината му е по-голяма от 199 преди да е въведен символът '.', вземат се първите 199 символа, свързват се с променливата s2 и изпълнението завършва. В противен случай, в s2 се записва редицата от символи до символа '.' (без него).

**Извеждане на низ**

Реализира се също по стандартния начин – чрез оператора cout.

*Пример:*

Операторът

cout << s;

извежда низа, свързан със s. Не е нужно да се грижим за дължината му. Знакът за край на низ идентифицира края на му.

**Дължина на низ**

Намира се чрез функцията strlen.

*Синтаксис*

strlen(<str>)

където

<str> е произволен низ.

*Семантика*

Намира дължината на <str>.

*Пример:*

strlen(“abc”) намира 3, а strlen(“”) - 0.

За използване на тази функция е необходимо да се включи заглавният файл string.h.

**Конкатенация на низове**

Реализира се чрез функцията strcat.

*Синтаксис*

strcat(<var_str>, <str>)

където

- <var_str> е променлива от тип низ;

<str> е низ (константа, променлива или по-общо израз със стойност символен низ).

*Семантика*

Конкатенира низа от <var_str> с низа <str>. Резултатът от конкатенацията се връща от функцията, а също се съдържа в променливата <var_str>. За използване на тази функция е необходимо да се включи заглавният файл string.h.

*Пример:*

```
#include <iostream.h>
#include <string.h>
int main()
{char a[10];
 cout << "a= ";
 cin >> a;     // въвеждане на стойност на a
 char b[4];
 cout << "b= ";
 cin >> b;     // въвеждане на стойност на b
 strcat(a, b);    // конкатениране на a и b, резултатът е в a
```

```
 cout << a << '\n';  // извеждане на a
 cout << strlen(strcat(a, b)) << '\n'; //повторна конкатенация
 return 0;
}
```

*Забележка:* Функцията strcat може да се използва и като оператор, и като израз. Обръщението към strcat от линия 10 е оператор, а това от линия 12 – израз от тип низ.

**Сравняване на низове**

Реализира се чрез функцията strcmp.

*Синтаксис*

strcmp(<str1>, <str2>)

където

<str1> и <str2> са низове (константи, променливи или по-общо изрази от тип низ).

*Семантика*

Низовете <str1> и <str2> се сравняват лексикографски. Функцията strcmp е целочислена. Резултатът от обръщението към нея е цяло число с отрицателна стойност (-1 за реализацията Visual C++ 6.0), ако <str1> е по-малък от <str2>, 0 – ако <str1> е равен на <str2> и с положителна стойност (1 за реализацията Visual C++ 6.0), ако <str1> е по-голям от <str2>.

За използване на strcmp е необходимо да се включи заглавният файл string.h.

*Примери:*

1. char a[10] = “qwerty”, b[15] = “qwerty”;
   if (!strcmp(a, b)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strcpm(a, b) връща 0 (низовете са равни), !strcmp(a, b) е 1 (true).

2. char a[10] = “qwe”, b[15] = “qwerty”;
   if (strcmp(a, b)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strcpm(a, b) връща -1 (низът a е по-малък от b).

3. char a[10] = “qwerty”, b[15] = “qwer”;
   if (strcmp(a, b)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strcpm(a, b) връща 1 (низът a е по-голям от b).

**Копиране на низове**

Реализира се чрез функцията strcpy.

*Синтаксис*

strcpy(<var_str>, <str>)

където

- <var_str> е променлива от тип низ;
- <str> е низ (константа, променлива или по-общо израз от тип низ).

*Семантика*

Копира <str> в <var_str>. Ако <str> е по-дълъг от допустимата за <val_str> дължина, са възможни труднооткриваеми грешки. Резултатът от копирането се връща от функцията, а също се съдържа в <var_str>.

За използване на тази функция е необходимо да се включи заглавният файл string.h.

*Пример:* Програмният фрагмент

```
char a[10];
strcpy(a, "1234567");
cout << a << "\n";
```

извежда

1234567

**Търсене на низ в друг низ**

Реализира се чрез функцията strstr.

*Синтаксис*

strstr(<str1>, <str2>)

където

<str1> и <str2> са произволни низове (константи, променливи или по-общо изрази).

*Семантика*

Търси <str2> в <str1>. Ако <str2> се съдържа в <str1>, strstr връща подниза на <str1> започващ от първото срещане на <str2> до края на <str1>. Ако <str2> не се съдържа в <str1>, strstr връща “нулев указател”. Последното означава, че в позиция на условие, функционалното обръщение ще има стойност false, но при други употреби (например извеждане), ще предизвиква грешка. Програмата ви ще извърши *нарушение при достъп* и ще блокира.

За използване на тази функция е необходимо да се включи заглавният файл string.h.

*Примери:* Програмният фрагмент

```
char str1[15] = "asemadaemada", str2[10]= "ema";
cout << strstr(str1, str2) << "\n";
```

извежда

```
emadaemada
```

а

```
char str1[15] = "asemadaemada", str2[10]= "ema";
cout << strstr(str2, str1) << "\n";
```

предизвиква съобщение за грешка по време на изпълнение.

**Преобразуване на низ в цяло число**

Реализира се чрез функцията atoi.

*Синтаксис*

atoi(<str>)

където <str> е произволен низ (константа, променлива или по-общо израз от тип низ).

*Семантика*

Преобразува символния низ <str> в число от тип int. Водещите интервали, табулации и знака за преминаване на нов ред се пренебрегват. Символният низ се сканира до първия символ различен от цифра. Ако низът започва със символ различен от цифра и знак, функцията връща 0.

За използване на тази функция е необходимо да се включи заглавният файл stdlib.h.

*Примери:*

Програмният фрагмент

```
char s[15] = "-123a45";
cout << atoi(s) << "\n";
```

извежда –123, а

```
char s[15] = "b123a45";
cout << atoi(s) << "\n";
```

извежда 0.

**Преобразуване на низ в реално число**

Реализира се чрез функцията atof.

*Синтаксис*

```
atof(<str>)
```

където <str> е произволен низ (константа, променлива или по-общо израз от тип низ).

*Семантика*

Преобразува символния низ в число от тип double. Водещите интервали, табулации и знакът за преминаване на нов ред се пренебрегват. Символният низ се сканира до първия символ различен от цифра. Ако низът започва със символ различен от цифра, знак или точка, функцията връща 0.

За използване на тази функция е необходимо да се включи заглавният файл stdlib.h.

*Примери:*

Програмният фрагмент

```
char s[15] = "-123.35a45";
cout << atof(st) << "\n";
```

извежда –123.35, а

```
char st[15] = ".123.34c35a45";
cout << atof(st) << "\n";
```

извежда 0.123.

**Допълнение:**

**Конкатенация на n символа от низ с друг низ**

Реализира се чрез функцията strncat.

*Синтаксис*

strncat(<var_str>, <str>, n)

където

- <var_str> е променлива от тип низ;
- <str> е низ (константа, променлива или по-общо израз от тип низ);
- n е цял израз с *неотрицателна* стойност.

*Семантика*

Копира първите n символа от <str> в края на низа, който е стойност на <var_str>. Копирането завършва когато са прехвърлени n символа, или е достигнат краят на <str>. Резултатът е в променливата <var_str>. За използване на тази функция е необходимо да се включи заглавният файл string.h.

*Пример:* Резултатът от изпълнението на фрагмента:

```
chat a[10] = “aaaaa”;
strncat(a, “qwertyqwerty”, 5);
cout << a;
```

е

aaaaaqwert

а на

```
strncat(a, “qwertyqwerty”, -5);
cout << a;
```

предизвиква съобщение за грешка (n е отрицателно).

**Копиране на n символа в символен низ**

Реализира се чрез функцията strncpy.

*Синтаксис*

strncpy(<var_str>, <str>, n)

където

- <var_str> е променлива от тип низ;
- <str> е низ (константа, променлива или по-общо израз);
- n е цял израз с *неотрицателна* стойност.

*Семантика*

Копира първите n символа на <str> в <var_str>. Ако <str> има по-малко от n символа, ‘\0’ се копира до тогава докато не се запишат n символа. Параметърът <var_str> трябва да е от вида char[m], m ≥ n, и съдържа резултатния низ. За използване на тази функция е необходимо да се включи заглавният файл string.h.

*Примери:* 1. Програмният фрагмент

```
char a[10];
strncpy(a, "1234567", 8);
cout << a << "\n";
```

извежда

```
1234567
```

Изпълнява се по следния начин: тъй като дължината на низа “1234567” е по-малка от 8, допълва се с един знак ‘\0’ и се свързва с променливата a.

2. Програмният фрагмент

```
char a[5];
strncpy(a, "123456789", 5);
cout << a << "\n";
```

извежда

```
12345<неопределено>
```

Изпълнява се по следния начин: тъй като дължината на низа “123456789” е по-голяма от 5, низът “12345” се свързва с променливата a, но не става допълване с ‘\0’, което личи по резултата.

**Сравняване на n символа на низове**

Реализира се чрез функцията strncmp.

*Синтаксис*

```
strncmp(<str1>, <str2>, n)
```

където

- <str1> и <str2> са низове (константи, променливи или по-общо изрази от тип ния);
- n е цял израз с *неотрицателна* стойност.

*Семантика*

Сравняват се лексикографски първите n символа на <str1> с низа <str2>. (Ако n е по-голямо от дължината на <str>, сравняват се лексикографски <str1> и <str2>). Резултатът от обръщението към функцията strncmp е цяло число с отрицателна стойност (-1 за Visual C++ 6.0), ако низът от първите n символа на <str1> е по-малък от <str2>, 0 – ако низът от първите n символа на <str1> е равен на <str2> и с положителна стойност (1 за Visual C++ 6.0), ако низът от първите n символа на <str1> е по-голям от <str2>.

За използване на strncmp е необходимо да се включи заглавният файл string.h.

*Примери:*

1. char a[10] = “qwer”, b[15] = “qwerty”;
   if (!strncmp(a, b, 3)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strncpm(a, b) връща 0 (низовете са равни), !strncmp(a, b) е 1 (true).

2. char a[10] = “qwer”, b[15] = “qwerty”;
   if (strncmp(a, b, 5)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strncpm(a, b) връща -1 (низът a е по-малък от b).

3. char a[10] = “qwerty”, b[15] = “qwer”;
   if (strncmp(a, b, 5)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strncpm(a, b) връща 1 (низът от първите 5 символа на a е по-голям от b).

**Търсене на символ в низ**

Реализира се чрез функцията strchr, съдържаща се в string.h.

*Синтаксис*

strchr(<str>, <expr>)

където

- <str> е произволен низ;
- <expr> е израз от интегрален или изброен тип с положителна стойност, означаваща ASCII код на символ.

*Семантика*

Търси първото срещане на символа, чийто ASCII код е равен на стойността на <expr>. Ако символът се среща, функцията връща подниза на <str> започващ от първото срещане на символа и продължаващ до края му. Ако символът не се среща - връща “нулев указател”. Последното означава, че в позиция на условие, функционалното обръщение ще има стойност false, но при други употреби (например извеждане), ще предизвиква грешка. Програмата ви ще извърши *нарушение при достъп* и ще блокира.

*Примери:* Операторът

cout << strchr(“qwerty”, ‘e’);

извежда

erty

Операторът

cout << strchr(“qwerty”, ‘p’);

предизвиква съобщение за грешка, а

if (strchr(“qwerty”, ‘p’)) cout << “yes \n”;
else cout << “no \n”;

извежда no, тъй като ‘p’ не се среща в низа “qwerty”.

**Търсене на първата разлика**

Реализира се чрез функцията strspn.

*Синтаксис*

strspn(<str1>, <str2>)

където <str1> и <str2> са произволни низове (константи, променливи или по-общо изрази от тип низ).

*Семантика*

Проверява до коя позиция <str1> и <str2> съвпадат. Връща дължината на низа до първия различен символ. За използване на тази функция е необходимо да се включи заглавният файл string.h.

*Пример:* Програмният фрагмент

```
char a[10]= "asdndf", b[15] = "asdsdfdhf";
cout << strspn(a, b) << "\n";
```

извежда 3 тъй като първият символ, по който се различават a и b е в позиция 4.

## Задачи върху тип низ

**Задача 61.** Да се напише програма, която проверява дали низ с дължина не по-голяма от 19 е палиндром, т.е. четен отляво надясно и отдясно наляво е един и същ.

Програма Zad61.cpp решава задачата. Тя обхожда в обратен ред символите на дадения низ a и ги записва в нов низ b (Ще отбележим, че присвояването b[n] = '\0'; е задължително). След това сравнява дадения с новия низ.

```
// Program Zad61.cpp
#include <iostream.h>
#include <string.h>
int main()
{char a[20], b[20];
 cout << "a= ";
 cin >> a;
 int n = strlen(a);
 int i;
 for (i = n-1; i >= 0; i--)
   b[n-i-1] = a[i];
 b[n] = '\0';
 if (!strcmp(a, b)) cout << "palindrome\n";
 else cout << "not palindrome \n";
 return 0;
}
```

**Задача 62.** Дадена е редица от n низа с дължина не по-голяма от 9. Да се напише програма, която конкатенира елементите на редицата.

Програма Zad62.cpp решава задачата.

```
// Program Zad62.cpp
#include <iostream.h>
#include <string.h>
int main()
{char a[20][10];  // a е масив от 20 низа char[10]
 cout << "n= ";
 int n;
 cin >> n;
 int i;
 for (i = 0; i <= n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
 char s[200] = ""; // s ще съдържа резултата от конкатенацията
 for (i = 0; i <= n-1; i++)
    strcat(s, a[i]);
 cout << s << '\n';
 return 0;
}
```

**Задача 63.** Дадена е редица от n низа (n > 1) с дължина не по-голяма от 9. Да се напише програма, която проверява дали редицата е монотонно намаляваща.

Програма Zad63.cpp решава задачата.

```
// Program Zad63.cpp
#include <iostream.h>
#include <string.h>
int main()
{char a[20][10];
 cout << "n= ";
 int n;
```

```
 cin >> n;
 int i;
 for (i = 0; i <= n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
 i = -1; int p;
 do
 {i++;
  p = strcmp(a[i], a[i+1]);
 } while ((p == 0 || p == 1) && i < n-2);
 if (p == 0 || p == 1) cout << "yes\n";
 else cout << "no\n";
 return 0;
}
```

**Задача 64.** Да се напише програма, която сортира във възходящ ред елементите на редица от низове, представящи имена не по-дълги от 9 знака. Сортираната редица да се изведе по 5 думи на ред.

Програма Zad64.cpp решава задачата.

```
// Program Zad64.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
int main()
{char a[100][10];
 cout << "n = ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error! \n";
  return 0;
 }
 int i;
 for (i = 0; i <= n-1; i++)
```

```
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
 for (i = 0; i <= n-2; i++)
 {char min[10];
  strcpy(min, a[i]);
  int k = i;
  for (int j = i+1; j <= n-1; j++)
    if (strcmp(a[j], min) == -1)
    {strcpy(min, a[j]);
     k = j;
    }
  strcpy(min, a[i]);
  strcpy(a[i], a[k]);
  strcpy(a[k], min);
 }
 for (i = 0; i <= n-1; i++)
 {cout << setw(10) << a[i];
  if (i != 0 && i % 5 == 0) cout << '\n';
 }
 cout << '\n';
 return 0;
}
```

**Задача 65.** Дадена е редица от n низа, представящи цели числа. Да се напише програма, която намира средно аритметичното на елементите на редицата.

Програма Zad65.cpp решава задачата.

```
// Program Zad65.cpp
#include <iostream.h>
#include <stdlib.h>
int main()
{char a[20][10];
 cout << "n= ";
 int n;
```

```
 cin >> n;
 for(int i = 0; i <= n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
 double average = 0;
 for (i = 0; i <= n-1; i++)
    average = average + atoi(a[i]);
 cout << average/n << '\n';
 return 0;
}
```

**Задача 66.** Да се напише програма, която въвежда квадратна матрица от низове с дължина не по-голяма от 9. Програмата да проверява дали матрицата е симетрична относно главния си диагонал.

Програма Zad66.cpp решава задачата.

```
// Program Zad66.cpp
#include <iostream.h>
#include <string.h>
int main()
{char a[20][20][10];
 cout << "n= ";
 int n;
 cin >> n;
 int i, j;
 for (i = 0; i <= n-1; i++)
    for (j = 0; j <= n-1; j++)
    {cout << "a[" << i << "][" << j << "]= ";
     cin >> a[i][j];
    }
 i = 0; int p;
 do
 {i++;
  j = -1;
```

```
     do
     {j++;
      p = !strcmp(a[i][j], a[j][i]);
     } while (p && j < i-1);
    } while (p && i < n-1);
    if (p) cout << "yes\n";
    else cout << "no \n";
    return 0;
   }
```

*Забележка:* В условията на циклите do/while е използван аритметичен израз на мястото на предикат (функцията strcmp е целочислена).

## Задачи

**Задача 1.** Да се напише програма, която намира скаларното произведение на реалните вектори $a = (a_0, a_1, ..., a_{n-1})$ и $b = (b_0, b_1,..., b_{n-1})$, $(1 \le n \le 50)$.

**Задача 2.** Да се напише програма, която въвежда n символа и намира и извежда минималния (максималния) от тях.

**Задача 3.** Да се напише програма, която:

а) въвежда редицата от n цели числа $a_0, a_1, ..., a_{n-1}$;

б) намира и извежда сумата на тези елементи на редицата, които се явяват удвоени нечетни числа.

**Задача 4.** Да се напише програма, която намира и извежда сумата от положителните и броя на отрицателните елементи на редицата от реални числа $a_0, a_1, ..., a_{n-1}$ $(1 \le n \le 30)$.

**Задача 5.** Да се напише програма, която изчислява реципрочното число на произведението на тези елементи на редицата $a_0, a_1, ..., a_{n-1}$ $(1 \le n \le 50)$, за които е в сила релацията $2 < a_i < i!$.

**Задача 6.** Да се напише програма, която изяснява, има ли в редицата от цели числа $a_0, a_1, ..., a_{n-1}$ $(1 \le n \le 100)$ два последователни нулеви елемента.

**Задача 7.** Дадени са сортираните във възходящ ред числови редици $a_0, a_1, ..., a_{k-1}$ и $b_0, b_1, ..., b_{k-1}$ $(1 \le k \le 40)$. Да се напише

програма, която намира броя на равенствата от вида $a_i = b_j$ ($i = 0, ..., k-1$, $j = 0, ..., k-1$).

**Задача 8.** Дадени са две редици от числа. Да се напише програма, която определя колко пъти първата редица се съдържа във втората.

**Задача 9.** Всяка редица от равни числа в едномерен сортиран масив, се нарича площадка. Да се напише програма, която намира началото и дължината на най-дългата площадка в даден сортиран във възходящ ред едномерен масив.

**Задача 10.** Да се напише програма, която по дадена числова редица $a_0, a_1, ..., a_{n-1}$ ($1 \le n \le 20$) намира дължината на максималната ѝ ненамаляваща подредица $a_{i1}, a_{i2}, ..., a_{iк}$ ($a_{i1} <= a_{i2} <= ... <= a_{iк}$, $i_1 < i_2 < ... < i_к$).

**Задача 11.** Дадена е редицата от символи $s_0, s_1, ..., s_{n-1}$ ($1 \le n \le 15$). Да се напише програма, която извежда отначало всички символи, които изобразяват цифри, след това всички символи, които изобразяват малки латински букви и накрая всички останали символи от редицата, запазвайки реда им в редицата.

**Задача 12.** Дадени са две цели числа, представени с масиви от символи. Да се напише програма, която установява дали първото от двете числа е по-голямо от второто.

**Задача 13.** Да се напише програма, която определя дали редицата от символи $s_0, s_1, ..., s_{n-1}$ ($1 \le n \le 45$) е симетрична, т.е. четена отляво надясно и отдясно наляво е една и съща.

**Задача 14.** Дадена е редицата от естествени числа $a_0, a_1, ..., a_{n-1}$ ($1 \le n \le 30$). Да се напише програма, която установява има ли сред елементите на редицата не по-малко от два елемента, които са степени на 2.

**Задача 15.** Дадени са полиномите Pn(x) и Qm(x). Да се напише програма, която намира:

а) сумата им;

б) произведението им.

**Задача 16.** За векторите $a = (a_0, a_1, ..., a_{n-1})$ и $b = (b_0, b_1, ..., b_{n-1})$ ($1 \le n \le 20$), да се определи дали са линейно зависими.

**Задача 17.** Дадена е квадратна целочислена матрица A с размерност nxn, елементите на която са естествени числа. Да се напише програма, която намира:

а) сумата от елементите под главния диагонал, които са прости числа;

б) произведението от елементите над главния диагонал, в записа на цифрите на които се среща цифрата 5;

в) номера на първия неотрицателен елемент върху главния диагонал.

**Задача 18.** Дадена е квадратната реална матрица A от n-ти ред. Да се напише програма, която намира:

а) сумата от елементите върху вторичния главен диагонал;

б) произведението от елементите под (над) вторичния главен диагонал.

**Задача 19.** Дадена е реална правоъгълна матрица A с размерност nxm. Да се напише програма, която изтрива k-тия ред (стълб) на A. Изтриването означава да се преместят редовете (стълбовете) с един нагоре (наляво) и намаляване броя на редовете (стълбовете) с един.

**Задача 20.** Върху равнина са дадени n точки чрез матрицата

$$x = \begin{pmatrix} x_{0,0} & x_{0,1} & \dots & x_{0,n-1} \\ x_{1,0} & x_{1,1} & \dots & x_{1,n-1} \end{pmatrix}$$

така, че $(x_{0,i}, x_{1,i})$ са координатите на i-тата точка. Точките по двойки са съединени с отсечки. Да се напише програма, която намира дължината на най-дългата отсечка.

**Задача 21.** Дадена е матрицата от цели числа

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \end{pmatrix}$$

Да се напише програма, която намира сумата на тези елементи $a_{1,i}$, $(0 \le i \le n-1)$, за които $a_{0,i}$ имат стойността на най-голямото сред елементите от първия ред на матрицата.

**Задача 22.** Дадено е множеството M от двойки

$$M = \{<x_0, y_0>, <x_1, y_1>, \dots, <x_{n-1}, y_{n-1}>\},$$

като $x_i$ и $y_i$ $(0 \le i \le n-1)$ са цели числа. Да се напише програма, която проверява дали множеството M дефинира функция.

*Упътване:* Множеството M дефинира функция, ако от $x_i = x_j$ следва $y_i = y_j$.

**Задача 23.** Дадена е целочислената квадратна матрица A от n-ти ред. Да се напише програма, която намира максималното от простите числа на A.

**Задача 24.** Казваме, че два стълба на една матрица си приличат, ако съвпадат множествата от числата, съставящи стълбовете. Да се напише програма, която намира номерата на всички стълбове на матрицата Anxm, които си приличат.

**Задача 25.** Матрицата A има седлова точка в $a_{i,j}$, ако $a_{i,j}$ е минимален елемент в i-я ред и максимален елемент в j-я стълб на A. Да се напише програма, която намира *всички* седлови точки на дадена матрица A.

**Задача 26.** Матрицата A има седлова точка в $a_{i,j}$, ако $a_{i,j}$ е минимален елемент в i-я ред и максимален елемент в j-я стълб на A. Да се напише програма, която установява дали *съществува* седлова точка в дадена матрица A.

**Задача 27.** Дадена е квадратна матрица A от n-ти ред. Да се напише програма, която установява дали съществува k ($0 \le k \le n-1$), така че k-я стълб на A (обхождан отгоре надолу) да съвпада с k-я й ред (обхождан отляво надясно).

**Задача 28.** Дадена е реалната квадратна матрица Anxn. Да се напише програма, която намира:

а) $\max\limits_{0\le i\le n-1} \{ \min\limits_{0\le j\le n-1} \{aij\}\}$

б) $\max\limits_{0\le j\le n-1} \{ \min\limits_{0\le i\le n-1} \{aij\}\}$

в) $\min\limits_{0\le i\le n-1} \{ \max\limits_{0\le j\le n-1} \{aij\}\}$

г) $\min\limits_{0\le j\le n-1} \{ \max\limits_{0\le i\le n-1} \{aij\}\}$

**Задача 29.** Дадена е квадратната матрица Anxn ($2 \le n \le 10$). Да се напише програма, която определя явява ли се A ортонормирана, т.е. такава, че скаларното произведение на всеки два различни реда на A е равно на 0, а скаларното произведение на всеки ред със себе си е равно на 1?

**Задача 30.** Да се напише програма, която определя явява ли се квадратната матрица Anxn магически квадрат, т.е. такава, че сумата от елементите от всички редове и стълбове е еднаква.

**Задача 31.** Дадена е система от линейни уравнения от n-ти ред. Да се напише програма, която я решава.

**Задача 32.** С тройката (i, j, v) се представя елемента v от i-я ред и j-я стълб на матрица. Две матрици с размерност nxn са представени като редици от тройки. Тройките са подредени по редове. Ако тройката (i, j, v) отсъства, приема се, че v = 0. Да се напише програма, която събира матриците и представя резултата като редица от тройки.

**Задача 33.** Да се напише програма, която сортира във възходящ ред елементите на всеки от редовете на квадратна матрица от низове, изразяващи думи с максимална дължина 9.

**Задача 34.** Дадена е квадратна матрица Anxn от низове, представящи думи с максимална дължина 9. Да се напише програма, която намира броя на палиндромите в частта над главния диагонал на A.

**Задача 35.** Дадена е квадратна матрица Anxn от низове, представящи думи с максимална дължина 6. Да се напише програма, която намира колко пъти думата s се среща в частта под вторичния главен диагонал на A.

**Задача 36.** Дадена е квадратна матрица Anxn от низове, представящи думи с максимална дължина 6. Да се напише програма, която проверява дали думата s се среща в частта над вторичния главен диагонал на A.

**Задача 37.** Дадена е квадратна матрица Anxn от низове, съдържащи думи с максимална дължина 9. Да се напише програма, която проверява дали изречението, получено след конкатенацията на думите от главния диагонал съвпада с изречението, получено по аналогичен начин за думите от вторичния главен диагонал на A.

**Задача 38.** Дадена е квадратна матрица A от n-ти ред от низове, съдържащи думи с максимална дължина 9. Да се напише програма, която намира изречението, получено след обхождане на A по спирала, започвайки от горния ляв ъгъл.

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. К. Хорстман, Принципи на програмирането със C++, С., СОФТЕХ, 2000.
3. М. Тодорова, Програмиране на Паскал, Полипринт, София, 1993.
4. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.

# 7

# Типове указател и псевдоним

## 7.1 Тип указател

Променлива, това е място за съхранение на данни, което може да съдържа различни стойности. Идентифицира се с дадено от потребителя име (идентификатор). Има си и тип. Дефинира се като се указват задължително типът и името ѝ. Типът определя броя на байтовете, в които ще се съхранява променливата, а също и множеството от операциите, които могат да се изпълняват над нея. Освен това, с променливата е свързана и стойност – неопределена или константа от типа, от който е тя. Нарича се още **rvalue**. Мястото в паметта, в което е записана rvalue, се идентифицира с адрес, който се нарича **адрес на променливата** или **lvalue**. По-точно адресът е адреса на първия байт от множеството байтове, отделени за променливата.

*Пример:* Фрагментът

```
int i = 1024;
```

дефинира променлива с име i и тип int. Стойността ѝ (rvalue) е 1024. i именува място от паметта (lvalue) с размери 4 байта, като lvalue е адреса на първия байт на това място.

Намирането на адреса на дефинирана променлива става чрез унарния префиксен дясно-асоциативен оператор & (амперсанд). Приоритетът му е същия като на унарните оператори +, -, !, ++, -- и др. Фиг. 7.1 описва оператора.

**Оператор &**

*Синтаксис*

&<променлива>

където <променлива> е вече дефинирана променлива.

*Семантика*

Намира адреса на <променлива>.

Фиг. 7.1 Оператор &

*Пример:* &i е адреса на променливата i и може да се изведе чрез оператора cout << &i;

Операторът & не може да се прилага върху константи и изрази, т.е. &100 и &(i+5) са недопустими обръщения. Не е възможно също прилагането му и върху променливи от тип масив, тъй като те имат и смисъла на *константни* указатели.

Адресите могат да се присвояват на специален тип променливи, наречени променливи от тип указател или само указатели.

**Дефиниране**

Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен указател към T. T се нарича **указван тип** или **тип на указателя**.

*Примери:*

int* е тип указател към тип int;

enum {a, b, c}* е тип указател към тип enum {a, b, c}.

**Множество от стойности**

Състои се от адресите на данните от тип T, дефинирани в програмата, преди използването на T*. Те са константите на типа T*. Освен тях съществува специална константа с име NULL, наречена **нулев указател**. Тя може да бъде свързвана с всеки указател независимо от неговия тип. Тази константа се интерпретира като “сочи към никъде”, а в позиция на предикат е false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа T*, се нарича **променлива от тип T***, **променлива от тип указател към тип T** или само **указател към тип T**. Дефинира се по общоприетия начин. Фиг. 7.2 показва синтаксиса на дефиницията на променлива от тип указател.

**Дефиниция на променлива от тип указател**

T* <променлива> [= <стойност>]$_{опц}$
 {,<променлива> [= <стойност>]$_{опц}$}$_{опц}$|
T *<променлива> [= <стойност>]$_{опц}$
 {*<променлива> [= <стойност>]$_{опц}$}$_{опц}$;

където

- T е име или дефиниция на тип;
- <променлива> е идентификатор;
- <стойност> е шестнадесетично число, представляващо адрес на данна от тип T или NULL.

Фиг. 7.2 Дефиниция на променлива от тип указател

T определя типа на данните, които указателят адресира, а също и начина на интерпретацията им.

Забелязваме, че фрагментите

<променлива> [=<стойност>] и

*<променлива> [=<стойност>]

могат да се повтарят. За разделител се използва запетаята. В първия случай на дефиницията (фиг. 7.2) обаче има особеност – операторът * се свързва с T само за първата променлива. Дефиницията

T* a, b;

е еквивалентна на

T* a;
T b;

т.е. само променливата a е от тип указател към тип T.

*Примери:* Дефиницията

int *pint1, *pint2;

задава два указателя към тип int, а

```
int *pint1, pint2;
```

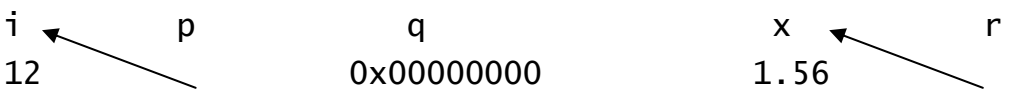
- указател pint1 към int и променлива pint2 от тип int.

Дефиницията на променлива от тип указател предизвиква в ОП да се отделят 4B, в които се записва някакъв адрес от множеството от стойности на съответния тип, ако дефиницията е с инициализация и неопределено или NULL, ако дефиницията не е с инициализация. (За реализацията Visual C++ 6.0 е неопределено). Този адрес е **стойността** на променливата от тип указател, а записаното на този адрес е **съдържанието** й.

*Пример:* В резултат от изпълнението на дефинициите

```
int i = 12;
int* p = &i;      // p е инициализирано с адреса на i
double *q = NULL; // q е инициализирано с нулевия указтел
double x = 1.56;
double *r = &x;   // r е инициализирано с адреса на x
```

ОП има вида:




**Съвет:** Всеки указател, който не сочи към конкретен адрес, е добре да се свърже с константата NULL. Ако по невнимание се опитате да използвате нулев указател, програмата ви може да извърши нарушение при достъп и да блокира, но това е по-добре, отколкото указателят да сочи към кой знай къде.

**Операции и вградени функции**

**Извличане на съдържанието на указател**

Осъществява се чрез префиксния, дясноасоциативен унарен оператор * (фиг. 7.3). * има приоритет на унарен оператор.

**Оператор ***

*Синтаксис*

*<променлива_от_тип_указател>*

*Семантика*

Извлича стойността на адреса, записан в <променлива_от_тип_ указател>, т.е. съдържанието на <променлива_от_тип_ указател>.

Фиг. 7.3 Оператор *

Като използваме дефинициите от примера по-горе, имаме:

*p е 12 // 12 е съдържанието на p

*r е 1.56 // 1.56 е съдържанието на r

Освен, че намира съдържанието на променлива от тип указател, обръщението

*<променлива_от_тип_указател>

е променлива от тип T. (Всички операции, допустими за типа T, са допустими и за нея.

Като използваме дефинициите от примера по-горе, *p и *r са цяла и реална променливи, съответно. След изпълнение на операторите за присвояване

*p = 20;

*r = 2.18;

стойността на i се променя на 20, а тази на r – на 2.18.

**Аритметични и логически операции**

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, >, >=, < и <=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още адресна аритметика. Особеността се изразява в т. нар. мащабиране. Ще го изясним чрез пример.

Да разгледаме фрагмента

```
int *p;
double *q;
...
p = p + 1;
q = q + 1;
```

Операторът p = p + 1; свързва p не със стойността на p, увеличена с 1, а с p + 1*4, където 4 е броя на байтовете, необходими за записване на данна от тип int (p е указател към int). Аналогично, q = q + 1; увеличава стойността на q не с 1, а с 8, тъй като q е указател към double (8 байта са необходими за записване на данна от този тип).

Общото правило е следното: Ако p е указател от тип T*, p+i е съкратен запис на p + i*sizeof(T), където sizeof(T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T.

**Въвеждане**

Не е възможно въвеждане на данни от тип указател чрез оператора cin. Свързването на указател със стойност става чрез инициализация или оператора за присвояване.

**Извеждане**

Осъществява се по стандартния начин - чрез оператора cout.

**Допълнение**

Типът, който се задава в дефиницията на променлива от тип указател, е информация за компилатора относно начина, по който да се интерпретира съдържанието на указателя. В контекста на последния пример *p са четири байта, които ще се интерпретират като цяло число от тип int. Аналогично, *q са осем байта, които ще се интерпретират като реално число от тип double.

Следващата програма илюстрира дефинирането и операциите за работа с указатели.

```
#include <iostream.h>
int main()
{int n = 10;   // дефинира и инициализира цяла променлива
 int* pn = &n; // дефинира и инициализира указател pn към n
 // показва, че указателят сочи към n
 cout << "n=  " << n << "  *pn= " << *pn << '\n';
```

```
 // показва, че адресът на n е равен на стойността на pn
 cout << "&n= "<< &n << "  pn= " << pn << '\n';
 // намиране на стойността на n чрез pn
 int m = *pn;  //  m == 10
 // промяна на стойността на n чрез pn
 *pn = 20;
 // извеждане на стойността на n
 cout << "n= " << n << '\n'; // n == 20
 return 0;
}
```

В някои случаи е важна стойността на променливата от тип указател (адресът), а не нейното съдържание. Тогава тя се дефинира като указател към *тип void*. Този тип указатели са предвидени с цел една и съща променлива - указател да може в различни моменти да сочи към данни от различен тип. В този случай, при опит да се използва съдържанието на променливата от тип указател, ще се предизвика грешка. Съдържанието на променлива - указател към тип void може да се извлече само след привеждане на типа на указателя (void*) до типа на съдържанието. Това може да се осъществи чрез операторите за преобразуване на типове.

*Пример:*

```
int a = 100;
void* p;                // дефинира указател към void
p = &a;                 // инициализира p
cout << *p;             // грешка
cout << *((int*) p);    // преобразува p в указател към int
                        // и тогава извлича съдържанието му.
```

В C++ е възможно да се дефинират указатели, които са константи (T* const <идентификатор>), а също и указатели, които сочат към константи (const T* <идентификатор>). И в двата случая се използва запазената дума const, която се поставя пред съответните елементи от дефинициите на указателите. Стойността на елемента, дефиниран като const (указателя или обекта, към който сочи) не може да бъде променяна. В дефиницията:

```
T* const <идентификатор>;
```

<идентификатор> е константен указател към тип T и не може да бъде променяна стойността му. В дефиницията:

```
const T* <идентификатор>;
```

<идентификатор> е указател към константа от тип T и не може да бъде променяно съдържанието му.

*Пример:*

```
int i, j = 5;
int *pi;              // pi е указател към int
int * const b = &i;   // b е константен указател към int
const int *c = &j;    // c е указател към цяла константа.
b = &j;               // грешка, b е константен указател
*c = 15;              // грешка, *c е константа
```

## 7.2 Указатели и масиви

В C++ има интересна и полезна връзка между указателите и масивите. Изразява се в това, че имената на масивите са указатели към техните "първи" елементи. Последното позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив.

*Указатели и едномерни масиви*

Нека a е масив, дефиниран по следния начин:

```
int a[100];
```

Тъй като a е указател към a[0], *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a + 1 е адреса на a[1], a + 2 е адреса на a[2] и т.н. a + n-1 е адреса на a[n-1]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, ..., n-1).

Има обаче една особеност. Имената на масивите са константни указатели. Заради това, някои от аритметичните оператори, приложими над указатели, не могат да се приложат над масиви. Такива са ++, -- и присвояването на стойност.

Следващата програма показва два начина за извеждане на елементите на масив.

```
#include <iostream.h>
int main()
{int a[] = {1, 2, 3, 4, 5, 6};
 int i;
 for (i = 0; i <=  5; i++)
    cout << a[i] << '\n';
 for (i = 0; i <=  5; i++)
    cout << *(a+i) << '\n';
 return 0;
}
```

Операторът

```
for (i = 0; i <= 5; i++)
{cout << *a << '\n';
 a++;
}
```

съобщава за грешка заради a++ (a е константен указател и не може да бъде променян). Може да се поправи като се използва помощна променлива от тип указател към int, инициализирана с масива a, т.е.

```
int* p = a;
for (i = 0; i <= 5; i++)
{cout << *p << '\n';
 p++;
 }
```

Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се преобразуват в *(a+i), т.е. операторът за индексиране [] се обработва от компилатора чрез адресна аритметика. Полезно е да отбележим, че операторът [] е ляво-асоциативен и с по-висок приоритет от унарните оператори (в частност от оператора за извличане на съдържание *).

**Указатели и двумерни масиви**

Името на двумерен масив е константен указател към първия елемент на едномерен масив от константни указатели. Ще изясним с пример казаното.

Нека a е двумерен масив, дефиниран по следния начин:

int a[10][20];

Променливата a е константен указател към първия елемент на едномерния масив a[0], a[1], ..., a[9], като всяко a[i] е константен указател към a[i][0] (i = 0,1, ..., 9), т.е.

```
a
↓
a[0] ──→ a[0][0]    a[0][1]    …       a[0][19]

a[1] ──→ a[1][0]    a[1][1]    …       a[1][19] …

                                …

a[9] ──→ a[9][0]    a[9][1]    …       a[9][19]
```

Тогава

**a == a[0][0]

a[0] == *a  a[1] == *(a + 1)  ...  a[9] == *(a + 9),

т.е.

a[i] == *(a + i)

Като използваме, че операторът за индексиране е лявоасоциативен и с по-висок приоритет от оператора *, получаваме:

a[i][j] == (*(a+i))[j] == *(*(a+i)+j).

**Задача 67.** Да се напише програма, която въвежда по редове правоъгълна матрица Anxk от цели числа. Конструира матрица, образувана от редовете на A от четна позиция, след което я извежда като увеличава с 1 всеки неин елемент. Конструира матрица, образувана от редовете на A от нечетна позиция, след което я извежда като увеличава с 2 всеки неин елемент. И накрая, ако n е четно,

извежда сумата на матриците от редовете от четните и нечетните позиции на A (без увеличенията с 1 и 2).

Програма Zad67.cpp решава задачата. Матрицата от редовете на A от четна позиция е конструирана в *масива от указатели p*, а матрицата от редовете на A от нечетна позиция е конструирана в *масива от указатели q*. Конструирането на масива p е реализирано чрез фрагмента:

```
int m = -1;
  for (i = 0; i <= n-1; i = i+2)
  {m++;
   *(p+m) = *(a+i);
  }
```

Стойността на p[0] е адреса на a[0][0], стойността на p[1] е адреса на a[2][0], стойността на p[2] е адреса на a[4][0] и т.н. Масивът q е конструиран по аналогичен начин, но стойността на q[0] е адреса на a[1][0], стойността на q[1] е адреса на a[3][0], стойността на q[2] е адреса на a[5][0] и т.н. Елементите на масива A не се копират на друго място в паметта. Така се прави икономия на ОП.

```
// Program Zad67.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[20][100];
 int* p[20];
 int* q[20];
 cout << "n, k = ";
 int n, k;
 cin >> n >> k;
 if (!cin)
 {cout << "Error! \n";
  return 0;
 }
 // въвеждане на матрицата
 int i, j;
 for (i = 0; i <= n-1; i++)
```

```
 for (j = 0; j <= k-1; j++)
 {cout << "a[" << i << "][" << j << "]= ";
  cin >> *(*(a + i) + j);
 }
// извеждане на матрицата
 for (i = 0; i <= n-1; i++)
 {for(j = 0; j <= k-1; j++)
   cout << setw(10) << *(*(a + i) + j);
  cout << '\n';
 }
 cout << "\n\n  Нова матрица от редовете в четни позиции \n";
 // конструиране
 int m = -1;
 for (i = 0; i <= n-1; i = i+2)
 {m++;
  *(p+m) = *(a+i);
 }
 // извеждане с увеличение на елементите с 1
 for (i = 0; i <= m; i++)
 {for(j = 0; j <= k-1; j++)
     cout << setw(10) << *(*(p + i) + j) + 1;
  cout << '\n';
 }
 cout << "\n\n  Нова матрица от редовете в нечетни позиции \n";
 // конструиране
 int l = -1;
 for (i = 1; i <= n - 1; i = i + 2)
 {l++;
  q[l] = a[i];
 }
 // извеждане с увеличение на елементите с 2
 for (i = 0; i <= l; i++)
 {for(j = 0; j <= k - 1; j++)
     cout << setw(10) << *(*(q + i) + j) + 2;
  cout << '\n';
 }
```

```
    // сумиране на двете матрици ако n е четно
    if (n%2 == 0)
     for (i = 0; i <= m; i++)
     {for (j = 0; j <= k-1; j++)
       cout << setw(10) << *(*(p + i) + j) + *(*(q + i) + j);
      cout << '\n';
     }
   return 0;
  }
```

## 7.3 Указатели и низове

Низовете са масиви от символи. Името на променлива от тип низ е константен указател, както и името на всеки друг масив. Така всичко, което казахме за връзката между масив и указател е в сила и за низ – указател.

Следващият пример илюстрира обхождане на низ чрез използване на указател към char. Обхождането продължава до достигане на знака за край на низ.

```
#include <iostream.h>
int main()
{char str[] = "C++Language"; // str е константен указател
 char* pstr = str;      // pstr е указател към низа str
 while (*pstr)
 {cout << *pstr << '\n';
  pstr++;
 }    // pstr вече не е свързан с низа “C++Language”.
 return 0;
}
```

Тъй като низът е зададен чрез масива от символи str, str е константен указател и не може да бъде променяна стойността му. Затова се налага използването на помощната променлива pstr. Ако низът е зададен чрез указател към char, както е в следващата програма, не се налага използването на такава.

```
#include <iostream.h>
int main()
```

```
{char* str = "C++Language";  // str е променлива
 while (*str)
 {cout << *str << '\n';
  str++;
 }
 return 0;
}
```

Примерите показват, че задаването на низ като указател към char има предимство пред задаването като масив от символи. Ще отбележим обаче, че дефиницията

```
char* str = "C++Language";
```

не може да бъде заменена с

```
char* str;
cin >> str;
```

следвани с въвеждане на низа “C++Language”(ще напомним, че не е възможно въвеждане на стойност на променлива от тип указател чрез оператора cin), докато дефиницията

```
char str[20];
```

позволява въвеждането му чрез cin, т.е.

```
cin >> str;
```

е допустимо.

Има още една особеност при дефинирането на низ като указател към char за реализацията Visual C++ 6.0. Ще я илюстрираме с пример.

Нека дефинираме променлива от тип низ по следния начин:

```
char s[] = “abba”;
```

Операторът

```
*s = ‘A’;
```

е еквивалентен на s[0] = ‘A’ и ще замени първото срещане на символа ‘a’ в s с ‘A’. Така

```
cout << s;
```

ще изведе низа

```
Abba
```

Да разгледаме съответната дефиницията на s чрез указател към char

```
char* s = “abba”;
```

Операторът

```
*s = ‘A’;
```

би трябвало да замени първото срещане на символа ‘a’ в s с ‘A’, тъй като s съдържа адреса на първото ‘a’. Тук обаче реализацията на Visual C++ съобщава за грешка – нарушение на достъпа. Последното може да се избегне като опцията на компилатора /ZI се замени с /Zi. Това се реализира като в менюто Project се избере Settings, след това C/C++, където Category трябва да има опция General. Накрая, в Project Options се промени /ZI на /Zi. Опцията /Zi се грижи за проблемите при нарушаване на достъпа - едно неудобство, което трябва да се има предвид.

## 7.4 Тип псевдоним

Чрез псевдонимите се задават алтернативни имена на обекти в общия смисъл на думата (променливи, константи и др.). В тази част ще ги разгледаме малко ограничено (псевдоними само за променливи).

**Дефиниране**

Нека T е име на тип. Типът T& е тип псевдоним на T. T се нарича **базов тип** на типа псевдоним.

**Множество от стойности**

Състои се от всички имена на дефинирани вече променливи от тип T.
*Пример:* Нека програмата съдържа следните дефиниции:

```
int a, b = 5;
...
int x, y = 9, z = 8;
...
```

Множеството от стойности на типа int& съдържа имената a, b, x, y, z. Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип псевдоним, се нарича променлива от този тип псевдоним. Фиг. 4 илюстрира дефиницията.

**Дефиниране на променлива от тип псевдоним**

<дефиниране_на_променлива_от_тип_псевдоним> ::=

T& <променлива> = <вече_дефинирана_променлива_от_тип_T>

{,<променлива> = <вече_дефинирана_променлива_от_тип_T>$_{опц}$};|

T &<променлива> = <вече_дефинирана_променлива_от_тип_T>

{, &<променлива> = <вече_дефинирана_променлива_от_тип_T>}$_{опц}$;

където T е име тип. Променливата след знак = се нарича инициализатор.

Фиг. 7.4 Дефиниране на променлива от тип псевдоним

Забелязваме, че е възможно фрагментите:

<променлива> = <вече_дефинирана_променлива_от_тип_T> и

&<променлива> = <вече_дефинирана_променлива_от_тип_T>

да се повтарят многократно. За разделител се използва символът запетая. В първият случай има една особеност. Дефиницията

T& a = b, c = d;

е еквивалентна на

T& a = b;

T c = d;

т.е. операторът & след типа T се отнася само за първата променлива след него.

*Пример:* Дефинициите

int a = 5;

int& syna = a;

double r = 1.85;

double &syn1 = r, &syn2 = r;

int& syn3 = a, syn4 = a;

определят syna и syn3 за псевдоними на цялата променлива a, syn1 и syn2 за псевдоними на реалната променлива r и syn4 за променлива от тип int.

**Дефиницията на променлива от тип псевдоним задължително е с инициализация – дефинирана променлива от същия тип като на базовия тип на типа псевдоним.** След това не е възможно променливата-псевдоним да стане псевдоним на нова променлива. Затова тя е “най-константната” променлива, която може да съществува.

*Пример:*

```
 ...
 int a = 5;
 int &syn = a;  // syn е псевдоним на a
 int b = 10;
 int& syn = b;    // error, повторна дефиниция
 ...
```

**Операции и вградени функции**

Дефиницията на променлива от тип псевдоним свързва променливата-псевдоним с инициализатора и всички операции и вградени функции, които могат да се прилагат над инициализатора, могат да се прилагат и над псевдонима му и обратно.

*Примери:*

```
1. int ii = 0;
   int& rr = ii;
   rr++;
   int* pp = &rr;
```

Резултатът от изпълнението на първите два оператора е следния:

ОП ii, rr

| | | |
|---|---|---|
| ... | 0 | ... |

Операторът rr++; не променя адреса на rr, а стойността на ii и тя от 0 става 1. В случая rr++ е еквивалентен на ii++.

Адресът на rr е адреса на ii. Намира се чрез &rr. Чрез дефиницията

```
int* pp = &rr;
```

pp е определена като указател към int, инициализирана с адреса на rr.

```
2. int a = 5;
   int &syn = a;
   cout << syn << "  " << a << '\n';
   int b = 10;
   syn = b;
```

```
      cout << b << "  " << a << “  “ << syn << '\n';
```
извежда
```
   5   5
   10  10   10
```
Операторът syn = b; е еквивалентен на a = b;.

```
   3. int i = 1;
      int& r = i; // r и i са свързани с 1
      cout << r; // извежда 1
      int x = r;  // x има стойност 1
      r = 2;   // еквивалентно е на i = 2;
```

*Допълнение:* Възможно е типът на инициализатора да е различен от този на псевдонима. В този случай се създава нова, наречена **временна**, променлива от типа на псевдонима, която се инициализира със зададената от инициализатора стойност, преобразувана до типа на псевдонима. Например, след дефиницията

```
   double x = 12.56;
   int& synx = x;
```
имаме

| ОП | x | | synx | |
|---|---|---|---|---|
| | 12.56 | ... | 12 | |
| | 8в | | 4в | |

Сега x и псевдонимът й synx са различни променливи и промяната на x няма да влияе на synx и обратно.

**Константни псевдоними**

В C++ е възможно да се дефинират псевдоними, които са константи. За целта се използва запазената дума const, която се поставя пред дефиницията на променливата от тип псевдоним. По такъв начин псевдонимът не може да променя стойността си, но ако е псевдоним на променлива, промяната на стойността му може да стане чрез промяна на променливата.

*Пример:* Фрагментът

```
int i = 125;
const int& syni = i;
cout << i << " " << syni << '\n'; // i и syni имат стойност 125
syni = 25;
cout << i << " " << syni << '\n';
```

ще съобщи за грешка (syni е константа и не може да е лява страна на оператор за присвояване), но фрагментът

```
int i = 125;
const int& syni = i;
cout << i << " " << syni << '\n';
i = i + 25;
cout << i << " " << syni << '\n';
```

ще изведе

```
125 125
150 150
```

Последното показва, че константен псевдоним на променлива защитава промяната на стойността на променливата чрез псевдонима.

## Допълнителна литература


1. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
2. B. Stroustrup, C++ Programming Language. Third Edition, Addison-Wesley, 1997.

# 8

# Функции

Добавянето на нови оператори и функции в приложенията, реализирани на езика C++, се осъществява чрез функциите. Те са основни структурни единици, от които се изграждат програмите на езика. Всяка функция се състои от множество от оператори, оформени подходящо за да се използват като обобщено действие или операция. След като една функция бъде дефинирана, тя може да бъде изпълнявана многократно за различни входни данни.

Програмите на езика C++ се състоят от една или повече функции. Сред тях задължително трябва да има точно една с име main и наречена **главна функция**. Тя е първата функция, която се изпълнява при стартиране на програмата. Главната функция от своя страна може да се обръща към други функции. Нормалното изпълнение на програмата завършва с изпълнението на главната функция (Възможно е изпълнението да завърши принудително с изпълнението на функция, различна от главната).

Използването на функции има следните предимства:

- Програмите стават ясни и лесни за тестване и модифициране.

- Избягва се многократното повтаряне на едни и същи програмни фрагменти. Те се дефинират еднократно като функции, след което могат да бъдат изпълнявани произволен брой пъти.

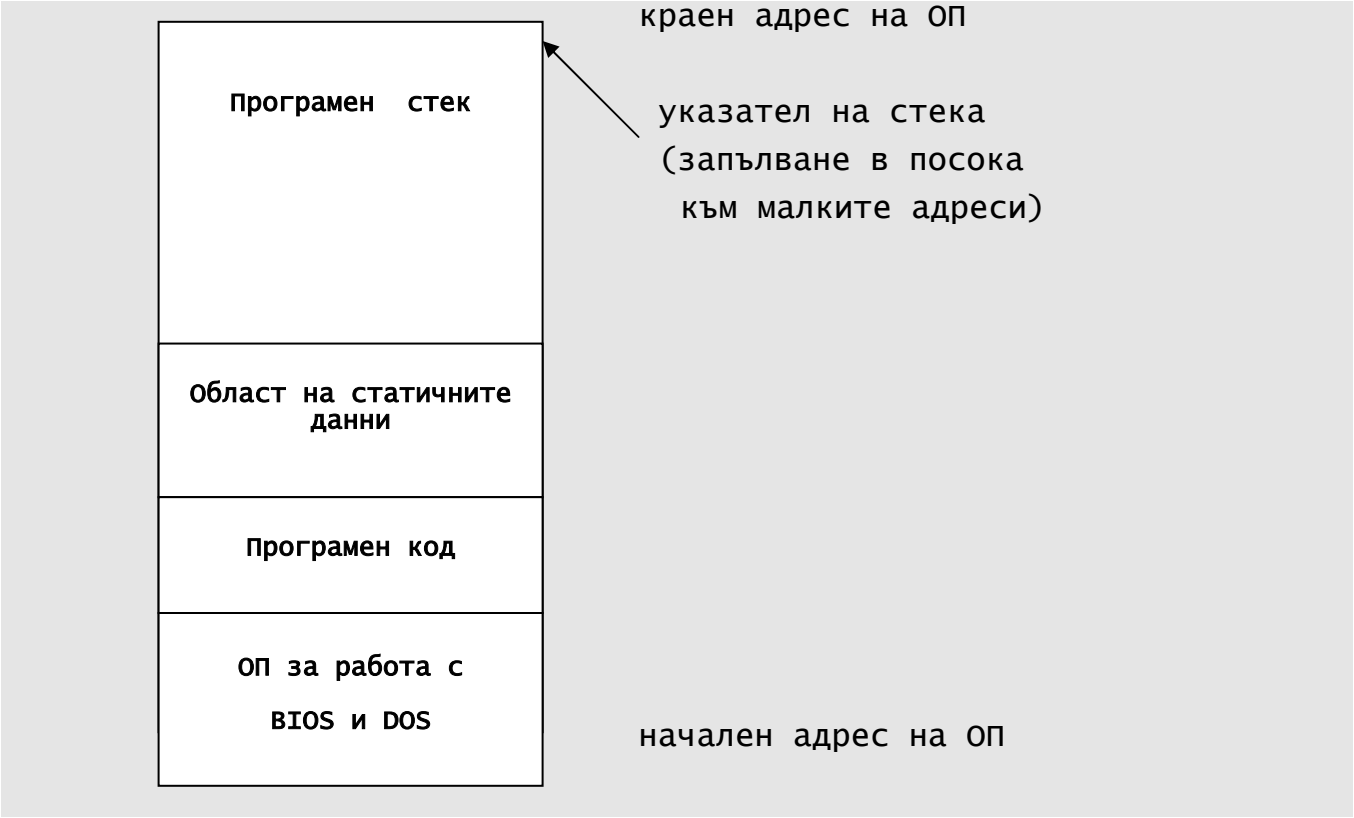
- Постига се икономия на памет, тъй като кодът на функцията се съхранява само на едно място в паметта, независимо от броя на нейните изпълнения.

Ще разгледаме най-общо разпределението на оперативната памет за изпълнима програма на C++. Чрез няколко примерни програми ще покажем

дефинирането, обръщението и изпълнението на функции, след което ще направим съответните обобщения.

## 8.1 Разпределение на ОП за изпълнима програма

Разпределението на ОП зависи от изчислителната система, от типа на операционната система, а също от модела памет. Най-общо се състои от: *програмен код, област на статичните данни, област на динамичните данни* и *програмен стек* (фиг. 8.1).




Фиг. 8.1 Разпределение на ОП

*Програмен код*

В тази част е записан изпълнимият код на всички функции, изграждащи потребителската програма.

*Област на статичните данни*

В нея са записани глобалните обекти (в широкия смисъл на думата) на програмата.

*Област на динамичните данни*

За реализиране на динамични структури от данни (списъци, дървета, графи, ...) се използват средства за динамично разпределение на паметта. Чрез тях се заделя и освобождава памет в процеса на изпълнение на програмата, а не преди това (при компилирането й). Тази памет е от областта на динамичните данни.

*Програмен стек*

Този вид памет съхранява данните на функциите на програмата. Стекът е динамична структура, организирана по правилото "последен влязъл – пръв излязъл". Той е редица от елементи с пряк достъп до елементите от единия си край, наречен **връх**. Достъпът се реализира чрез указател. Операцията включване се осъществява само пред елемента от върха, а операцията изключване – само за елемента от върха.

Елементите на програмния стек са "блокове" от памет, съхраняващи данни, дефинирани в някаква функция. Наричат се **стекови рамки**.

## 8.2 Примери за програми, които дефинират и използват функции

**Задача 68.** Да се напише програма, която въвежда стойности на естествените числа a, b, c и d и намира и извежда най-големият общ делител на числата a и b, след това на c и d и накрая на a, b, c и d.

Програма Zad68.cpp решава задачата. Тя се състои от две функции: gcd и main. Функцията gcd(x, y) намира най-големия общ делител на естествените числа x и y. Тъй като main се обръща към (извиква) gcd, функцията gcd трябва да бъде известна преди функцията main. Най-лесният начин да се постигне това е във файла, съдържащ програмата, първо да се постави дефиницията на gcd, а след това тази на main. Ще бъде показан алтернативен начин по-късно.

Описанието на функцията gcd прилича на това на функцията main. Състои се от заглавие

```
int gcd(int x, int y)
```

и тяло

```
{while (x != y)
 if (x > y) x = x-y; else y = y-x;
```

```
 return x;
 }
```

Заглавието определя, че gcd е име на двуаргументна целочислена функция с цели аргументи, т.е.

gcd: int x int $\longrightarrow$ int

Името е произволен идентификатор. В случая е направен мнемонически избор. Запазената дума int пред името на функцията е типа й (по-точно е типа на резултата на функцията). В кръгли скоби и отделени със запетая са описани параметрите x и y на gcd. Те са различни идентификатори. Предшестват се от типовете си. Наричат се **формални параметри за функцията**.

Тялото на функцията е блок, реализиращ алгоритъма на Евклид за намиране на най-големия общ делител на естествените числа x и y. Завършва с оператора

return x;

чрез който се прекратява изпълнението на функцията като стойността на израза след return се връща като стойност на gcd в мястото, в случая в main, в което е направено обръщението към нея.

```
// Program Zad68.cpp
#include <iostream.h>
int gcd(int x, int y)
{while (x != y)
 if (x > y) x = x-y; else y = y-x;
 return x;
 }
 int main()
 {cout << "a, b, c, d= ";
  int a, b, c, d;
  cin >> a >> b >> c >> d;
  if (!cin || a < 1 || b < 1 || c < 1 || d < 1)
  {cout << "Error! \n";
   return 1;
  }
  int r = gcd(a, b);
  cout << "gcd{" << a << ", " << b << "}= " << r << "\n";
```

```
  int s = gcd(c, d);
  cout << "gcd{" << c << ", " << d << "}= " << s << "\n";
  cout << "gcd{" << a << ", " << b << ", " << c << ", "
       << d << "}= " << gcd(r, s) << "\n";
   return 0;
 }
```

*Изпълнение на програма zad68.cpp*

Дефинициите на функциите main и gcd се записват в областта на паметта, определена за програмния код. Изпълнението на програмата започва с изпълнение на функцията main. Фрагментът

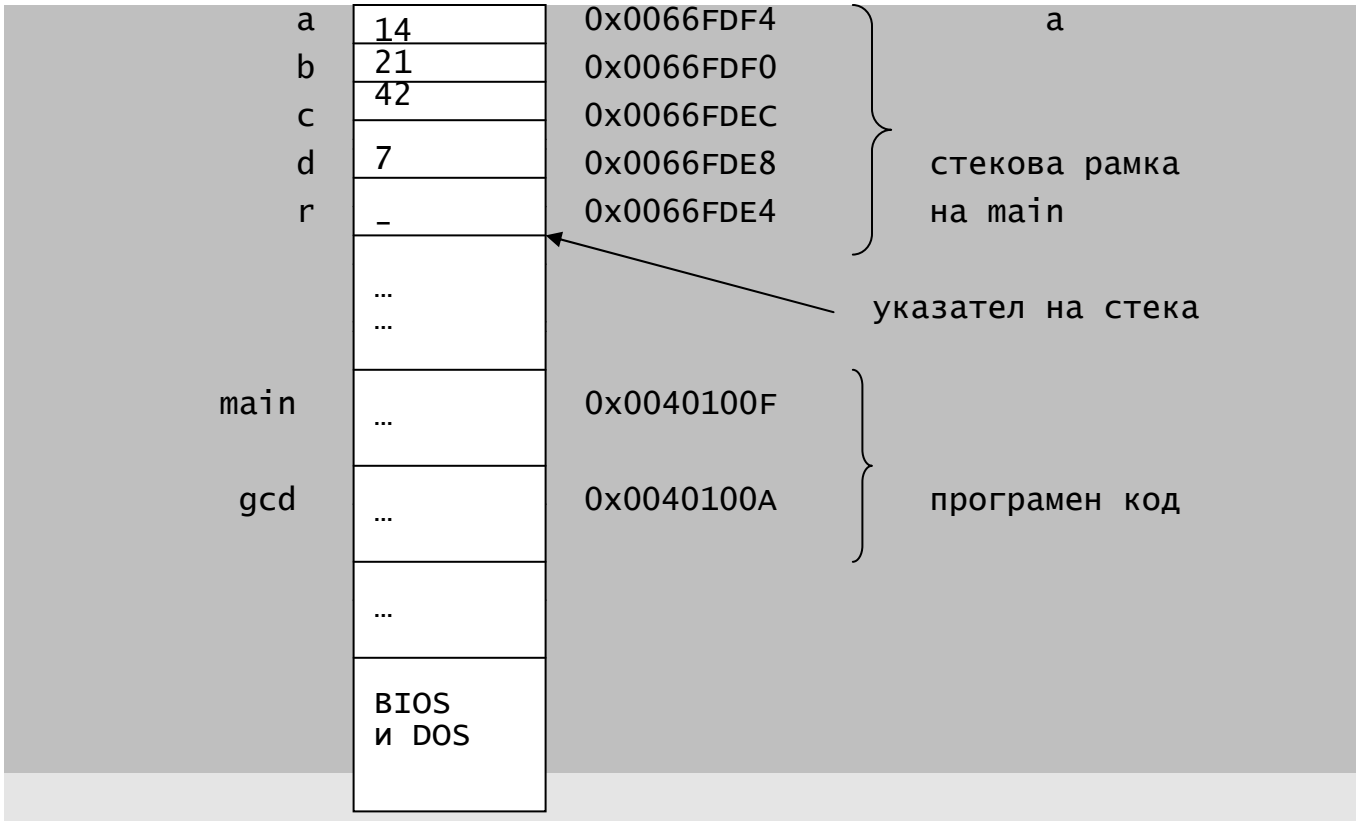
```
cout << "a, b, c, d= ";
int a, b, c, d;
cin >> a >> b >> c >> d;
if (!cin || a < 1 || b < 1 || c < 1 || d < 1)
{cout << "Error \n";
 return 1;
}
```

дефинира и въвежда стойности на целите променливи a, b, c и d като осигурява да са естествени числа. Нека за a, b, c и d са въведени 14, 21, 42 и 7 съответно. В тази последователност те се записват в дъното на програмния стек (Фиг. 8.2). Така на дъното на стека се оформя “блок” от памет за main с достатъчно големи размери, който освен променливите от main съдържа и някои “вътрешни” данни. Този блок се нарича **стекова рамка на main**.

Операторът

```
int r = gcd(a, b);
```

дефинира цялата променлива r като в стековата рамка на main, веднага след променливата d отделя 4B, в които ще запише резултатът от обръщението gcd(a, b) към функцията gcd. Променливите a и b се наричат **фактически параметри за това обръщение**. Забелязваме, че типът им е същия като на съответните им формални параметри x и y.




Фиг. 8.2 Разпределение на ОП програмата Zad68.cpp

*Обръщение към gcd(a, b)*

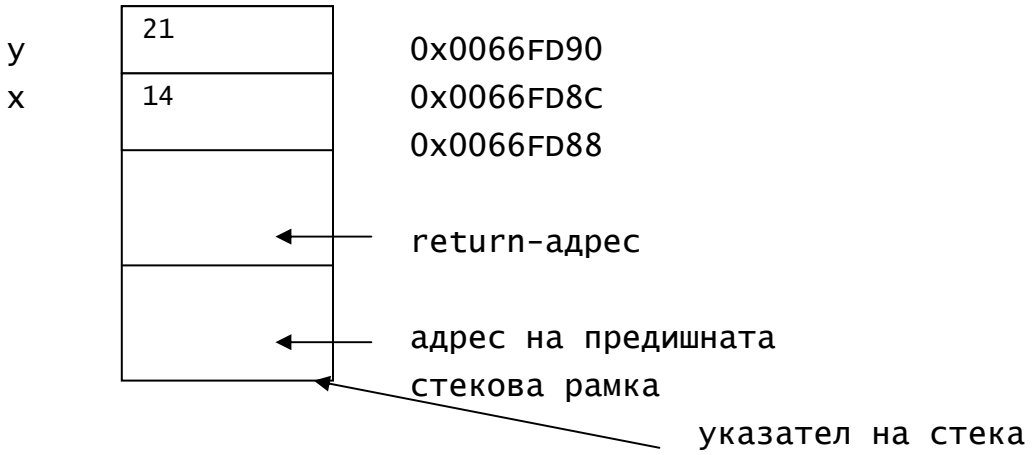
В програмния стек се генерира нов блок памет – стекова рамка за функцията gcd. В него се записват формалните и локалните параметри на gcd, а също и някои “вътрешни” данни като return-адреса и адреса на стековата рамка на main. Указателят на стека се премества след тази стекова рамка.

Обръщението се осъществява по следния начин:

*а) Свързване на формалните с фактическите параметри*

В стековата рамка на gcd, се отделят по 4 байта за формалните параметри x и y в обратен ред на реда, в които са записани в заглавието. В тази памет се **откопирват стойностите** на съответните им фактически параметри. Отделят се също 4B за т. нар. return-адрес, адреса на мястото в main, където ще се върне резултатът, а също се отделя памет, в която се записва адресът на предишната стекова рамка, т.е.

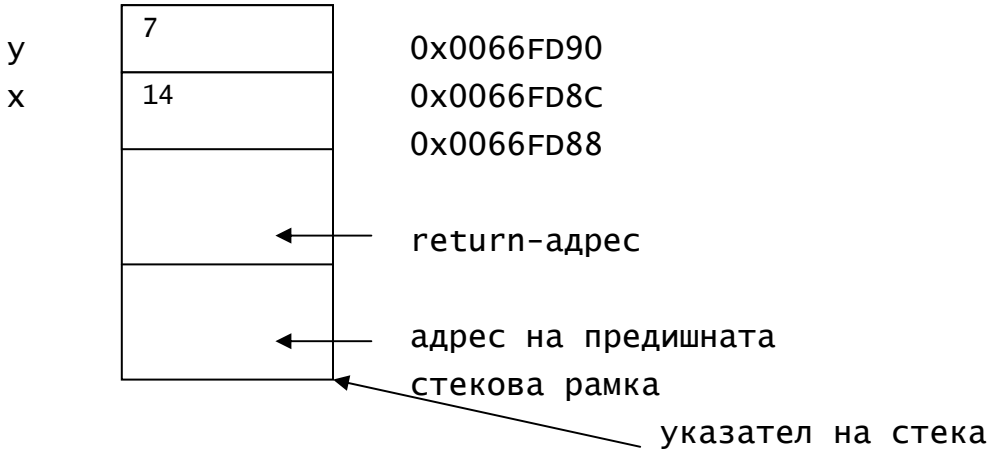
памет за gcd (I-во обръщение към него)




*б) Изпълнение на тялото на gcd*

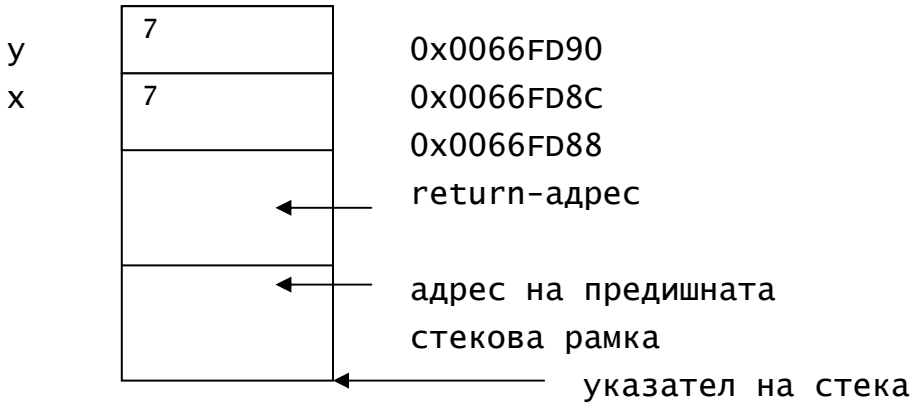
Тъй като е в сила y > x, стойността на y се променя на 7, т.е.

памет в стека за gcd




Сега пък е в сила x > y, което води до промяна на стойността на x на 7, т.е.

памет в стека за gcd



Операторът за цикъл завършва изпълнението си. Изпълнението на оператора

```
return x;
```

преустановява изпълнението на gcd като връща в main в мястото на прекъсването (return-адреса) стойността 7 на обръщението gcd(a, b). Отделената за gcd стекова рамка се освобождава. Указателят на стека се установява в края на стековата рамка на main. Изпълнението на програмата продължава с инициализацията на r. Резултатът от обръщението gcd(14, 21) се записва в отделената за r памет.

Операторът

```
cout << "gcd{" << a << ", " << b << "}= " << r << "\n";
```

извежда получения резултат.

Изпълнението на останалите обръщения към gcd се реализира по същия начин. При обръщението към всяко от тях в стека се създава стекова рамка на gcd, а след завършване на обръщението, рамката се освобождава. При достигане до оператора return 0; от main, се освобождава и стековата рамка на main.

Функцията gcd реализира най-простото и “чисто” дефиниране и използване на функции – получава входните си стойности единствено чрез формалните си параметри и връща резултата от работата си чрез оператора return. Забелязваме, че обръщението gcd(a, b) работи с копия на стойностите на a и b, запомнени в x и y, а не със самите a и b. В процеса на изпълнение на тялото на gcd, стойностите на x и y се променят, но това не оказва влияние на стойностите на фактическите параметри a и b.

Такова свързване на формалните с фактическите параметри се нарича **свързване по стойност** или още **предаване на параметрите по стойност**. При него фактическите параметри могат да бъдат не само променливи, но и изрази от типове, съвместими с типовете на съответните формални параметри. Обръщението gcd(gcd(a, b), gcd(c, d)) е коректно и намира най-големия общ делител на a, b, c и d.

В редица случаи се налага функцията да получи входа си чрез някои от формалните си параметри и да върне резултат не по обичайния начин – чрез оператора return, а чрез същите или други параметри. Задача 69 дава пример за това.

**Задача 69.** Да се напише програма, която въвежда стойности на реалните променливи a, b, c и d, след което разменя стойностите на a и b и на c и d съответно.

Ако дефинираме функция swapi(double* x, double* y), която разменя стойностите на реалните променливи, към които сочат указателите x и y, обръщението swapi(&a, &b) ще размени стойностите на a и b, а обръщението swapi(&c, &d) ще размени стойностите на c и d. Програма Zad69.cpp решава задачата. Тя се състои от функциите: swapi и main. Тъй като main се обръща към (извиква) swapi, функцията swapi трябва да бъде известна преди функцията main. Затова във файла, съдържащ програмата, първо е поставена функцията swapi, а след това - main.

```
// Program Zad69.cpp
#include <iostream.h>
#include <iomanip.h>
void swapi(double* x, double* y)
{double work = *x;
 *x = *y;
 *y = work;
 return;
}
int main()
{cout << "a, b, c, d= ";
 double a, b, c, d;
 cin >> a >> b >> c >> d;
 cout << setprecision(2) << setiosflags(ios :: fixed);
 cout << setw(10) << a << setw(10) << b
      << setw(10) << c << setw(10) << d << '\n';
 swapi(&a, &b);
 swapi(&c, &d);
 cout << setw(10) << a << setw(10) << b
      << setw(10) << c << setw(10) << d << '\n';
 return 0;
}
```

Функцията swapi има подобна структура като на gcd. Но и заглавието, и тялото й са по-различни. Типът на swapi е указан чрез запазената дума void. Това означава, че функцията не връща стойност чрез оператора return. Затова в тялото на swapi е пропуснат изразът след return (възможно е да бъде пропуснат и самия return). Формалните параметри x и y са указатели към типа double, а в тялото се работи със съдържанията на указателите.

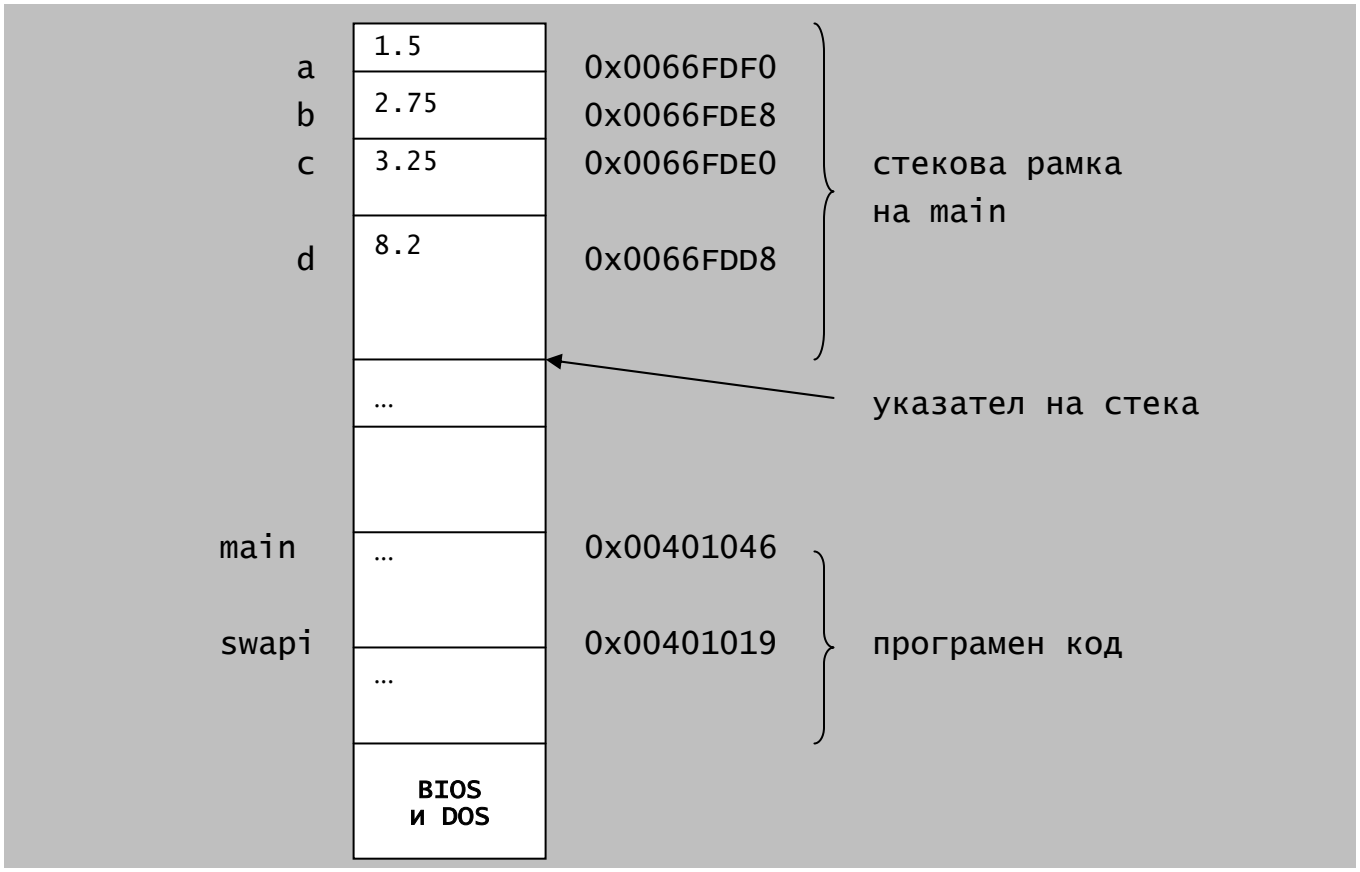
Забелязваме също, че обръщенията към swapi в main

```
swapi(&a, &b);
swapi(&c, &d);
```

не участват като аргументи на операции, а са оператори.

*Изпълнение на програма Zad69.cpp*

Дефинициите на функциите main и swapi се записват в областта на паметта, определена за програмния код. Изпълнението на програмата започва с изпълнение на функцията main.




Фиг. 8.3 Разпределение на паметта за swapi

Фрагментът

```
cout << "a, b, c, d= ";
double a, b, c, d;
cin >> a >> b  >> c >> d;
cout << setprecision(2) << setiosflags(ios :: fixed);
cout << setw(10) << a << setw(10) << b
     << setw(10) << c << setw(10) << d << '\n';
```

дефинира и въвежда стойности за реалните променливи a, b, c и d и ги извежда върху екрана според дефинираното форматиране. Нека за стойности на a, b, c и d са въведени 1.5, 2.75, 3.25 и 8.2 съответно (Фиг. 8.3).

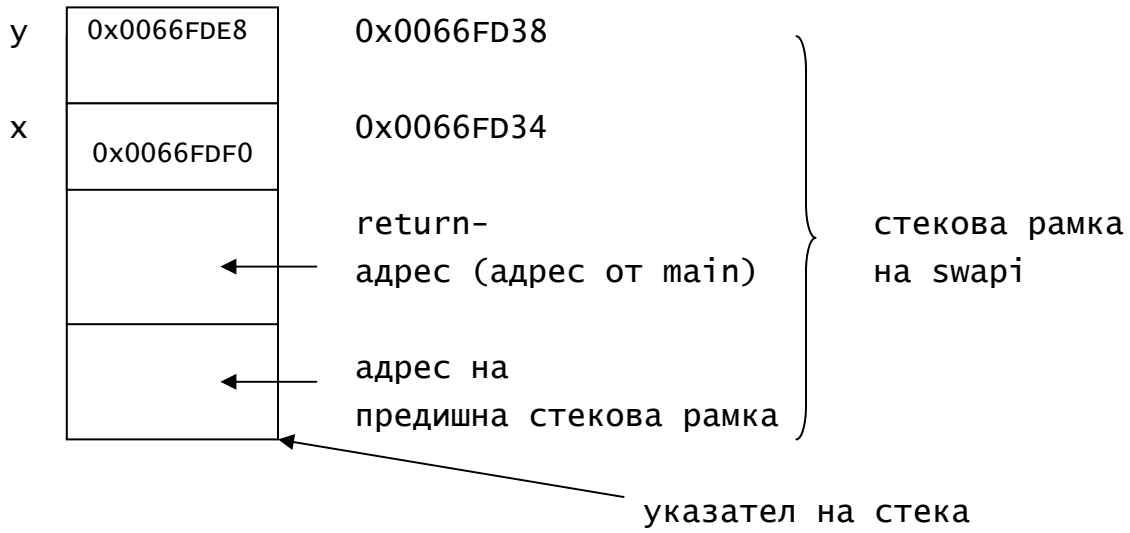
Обръщението

```
swapi(&a, &b);
```

се изпълнява по следния начин:

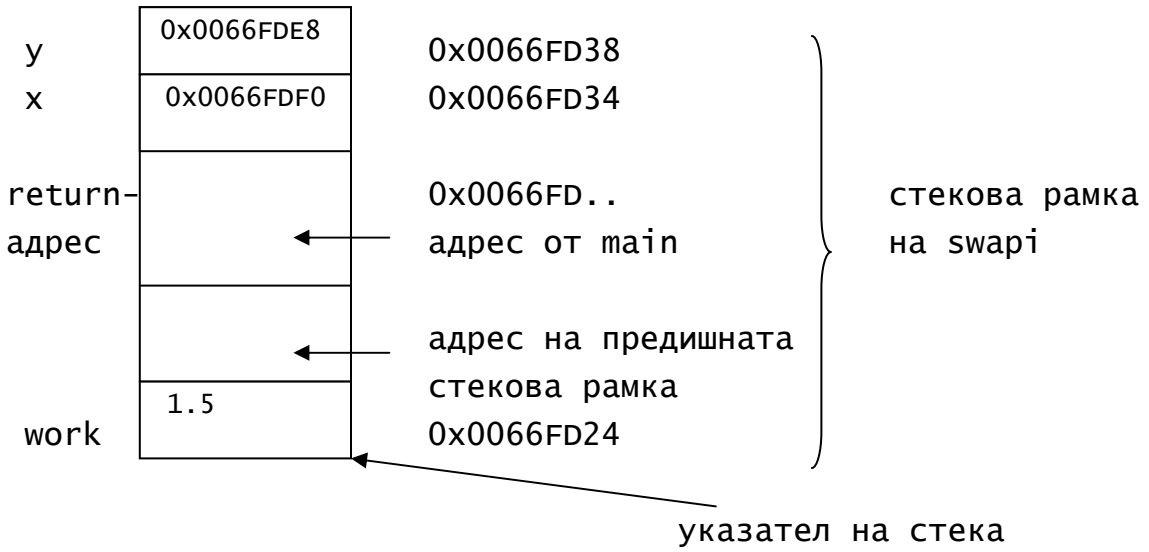
*а) Свързване на формалните с фактическите параметри*

В стека се конструира нова рамка – рамката на swapi. Указателят на стека се установява след нея. Отделят се по 4 байта за формалните параметри x и y, в която памет се **записват адресите** на съответните им фактически параметри, още 4B, в които се записва адресът на swapi(c, d), от където трябва да се продължи изпълнението на main (return-адреса), а също и памет, в която се записва адресът на предишната стекова рамка (в случая на main).




*б) Изпълнение на тялото на swapi*

Изпълнява се като блок. За реалната променлива work се отделят 8 байта в стековата рамка на swapi, в които се записва съдържанието на x, в случая 1.5, т.е.



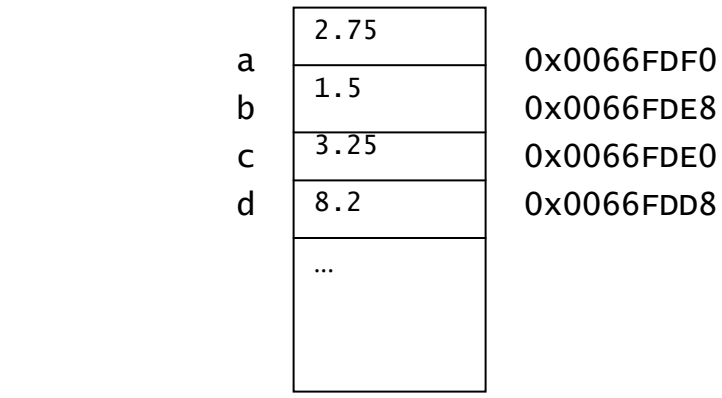

Операторът

```
*x = *y;
```

променя съдържанието на x с това на y, а операторът

```
*y = work;
```

променя съдържанието на y като го свързва със стойността на work, т.е.




Операторът return; прекъсва работа на на swapi и предава управлението в точката на извикването му в главната функция (return-адреса). Стековата рамка, отделена за swapi се освобождава. Указателят на

стека сочи стековата рамка на main. В резултат стойностите на променливите a и b са разменени.

Обръщението swapi(c, d) се изпълнява по аналогичен начин. За нея се генерира нова стекова рамка (на същите адреси), която се освобождава когато изпълнението на swapi завърши.

Функцията swapi получава входните си стойности чрез формалните си параметри и връща резултата си чрез тях. Забелязваме, че обръщението swapi(&a, &b) работи не с копия на стойностите на a и b, а с адресите им. В процеса на изпълнение на тялото се променят стойностите на фактическите параметри a и b при първото обръщение към нея и на c и d – при второто.

Такова свързване на формалните с фактическите параметри се нарича **свързване на параметрите по указател** или още **предаване на параметрите по указател** или **свързване по адрес**. При този вид предаване на параметрите, фактическите параметри задължително са променливи или адреси на променливи.

Освен тези два начина на предаване на параметри, в езика C++ има още един – **предаване на параметри по псевдоним**. Той е сравнително по-удобен от предаването по указател и се предпочита от програмистите.

Ще го илюстрираме чрез същата задача. Програма Zad69_1.cpp реализира функция swapi, в която предаването на параметрите е по псевдоним.

```
// Program Zad69_1.cpp
#include <iostream.h>
#include <iomanip.h>
void swapi(double& x, double& y)
{double work = x;
 x = y;
 y = work;
 return;
}
int main()
{cout << "a, b, c, d= ";
 double a, b, c, d;
 cin >> a >> b >> c >> d;
```

```
 cout << setprecision(2) << setiosflags(ios :: fixed);
 cout << setw(10) << a << setw(10) << b
      << setw(10) << c << setw(10) << d << '\n';
 swapi(a, b);
 swapi(c, d);
 cout << setw(10) << a << setw(10) << b
      << setw(10) << c << setw(10) << d << '\n';
 return 0;
}
```

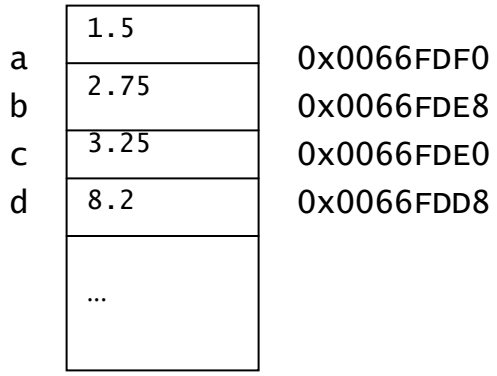
Ще проследим изпълнението и на тази модификация.

Изпълнението на програмата започва с изпълнение на функцията main. Фрагментът

```
cout << "a, b, c, d= ";
double a, b, c, d;
cin >> a >> b >> c >> d;
cout << setprecision(2) << setiosflags(ios :: fixed);
cout << setw(10) << a << setw(10) << b
     << setw(10) << c << setw(10) << d << '\n';
```

дефинира и въвежда стойности за реалните променливи a, b, c и d и ги извежда върху екрана според дефинираното форматиране. Нека за стойности на a, b, c и d отново са въведени 1.5, 2.75, 3.25 и 8.2 съответно. След обработката му в стека се конструира стековата рамка на main.

памет на main




Обръщението

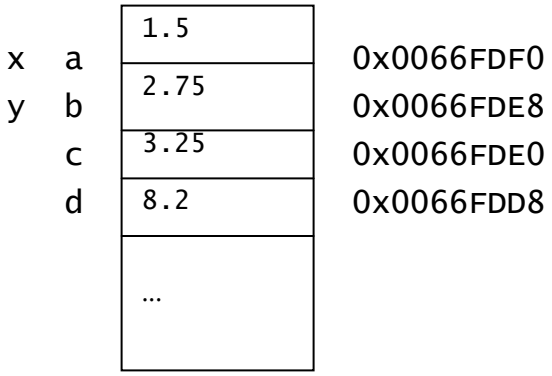
```
swapi(a, b);
```

се изпълнява по следния начин:

*а) Свързване на формалните с фактическите параметри*

За целта се генерира нова стекова рамка – рамката на swapi. Указателят на стека сочи тази рамка. Тъй като формалните параметри x и y са псевдоними на променливите a и b съответно, за тях памет в стековата рамка на swapi не се отделя. Параметърът x "прелита" и се "закачва" за фактическия параметър a и аналогично y "прелита" и се "закачва" за фактическия параметър b от стековата рамка на main. Така всички действия с x и y в swapi се изпълняват с фактическите параметри a и b от main съответно.

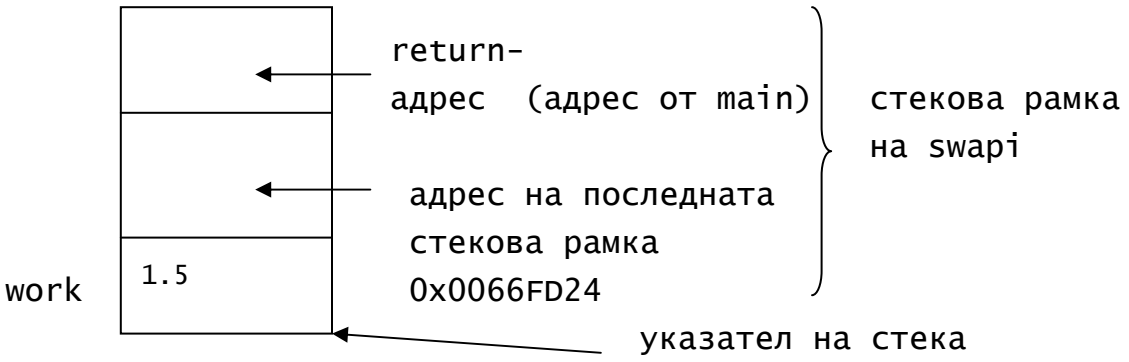
памет на main




*б) Изпълнение на тялото на swapi*

Изпълнява се като блок. В рамката на swapi, за реалната променлива work се отделят 8 байта, в които се записва стойността на x, в случая 1.5, т.е.

стекова рамка на swapi




Операторът

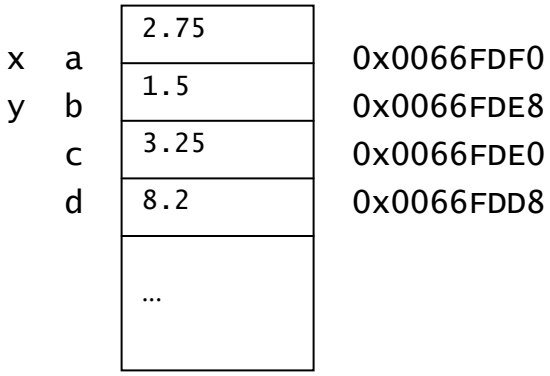
x = y;

присвоява на a стойността на b, а операторът

```
y = work;
```

променя стойността на променливата b като й присвоява стойността на work, т.е.

стекова рамка на main




Операторът return; прекъсва работа на на swapi и предава управлението на return-адреса от главната функция. Стековата рамка на swapi се освобождава. Указателят на стека сочи стековата рамка на main. В резултат, стойностите на променливите a и b са разменени. Променливите a и b са “освободени от“ x и y. Следва изпълнение на обръщението

```
swapi(c, d);
```

което се реализира по същия начин (даже стековата му рамка се разполага на същите адреси в стека).

Забелязваме, че фактическите параметри, съответстващи на формални параметри-псевдоними са променливи.

Тази реализация на swapi е по-ясна и удобна от съответната й с указатели. Тялото й реализира размяна на стойностите на две реални променливи без да се налага използването на адреси.

Нека в тялото на main на Zad69_1.cpp преди оператора return; включим фрагмента:

```
int m, n;
cin >> m >> n;
swapi(m, n);
cout << setw(10) << m << setw(10) << n << "\n";
```

Ще отбележим, че m и n са от тип int, а формалните параметри на swapi са псевдоними на тип double. Някои реализации (в това число Visual

C++ 6.0) ще сигнализират грешка на третата линия – невъзможност за преобразуване на параметър от int в double &, други обаче ще имат нормално поведение, но няма да разменят стойностите на m и n. Последното е така, тъй като при несъответствие на типа на псевдонима с типа на инициализатора, в стековата рамка на swapi, се създават "временни" променливи x и y, в които се запомнят конвертираните стойности на инициализаторите. Размяната се извършва, но само в стековата рамка на swapi.

При предаване на параметрите по псевдоним или по указател, фактическите параметри са променливи или адреси на променливи, за разлика от предаването на параметри по стойност, когато фактическите параметри могат да са изрази в общия случай.

Възможно е някои параметри да се подават по стойност, други по псевдоним или по указател, а също функцията да връща резултат и чрез оператора return. Примери ще бъдат дадени в следващите части на изложението. Ще бъдат обсъдени също предимствата и недостатъците на всеки от начините за предаване на параметрите.

Ако функция не връща резултат чрез return (типът й е void), се нарича още **процедура**.

Разгледаните програми се състояха от две функции. По-сериозните приложения съдържат повече функции. Подредбата им може да започва с main, след която в произволен ред да се дефинират останалите функции. В този случай, дефиницията на main трябва да се предшества от *декларациите* на останалите функции. Декларацията на една функция се състои от заглавието й, следвано от ;. Имената на формалните параметри могат да се пропуснат. Например, програмата от Zad69_1.cpp може да се запише във вида:

```
// Program Zad69_1.cpp
#include <iostream.h>
#include <iomanip.h>
void swapi(double&, double&); // декларация на swapi
int main()
{cout << "a, b, c, d= ";
 double a, b, c, d;
 cin >> a >> b >> c >> d;
```

```
 cout << setprecision(2) << setiosflags(ios :: fixed);
 cout << setw(10) << a << setw(10) << b
      << setw(10) << c << setw(10) << d << '\n';
 swapi(a, b);
 swapi(c, d);
 cout << setw(10) << a << setw(10) << b
      << setw(10) << c << setw(10) << d << '\n';
 return 0;
}
void swapi(double& x, double& y) // дефиниция на swapi
{double work = x;
 x = y;
 y = work;
 return;
}
```

## 8.3 Дефиниране на функции

**Синтаксис**

Дефиницията на функция се състои от две части: заглавие (прототип) и тяло. Синтаксисът й е показан на фиг. 8.4.

**Дефиниране на функция**

[<модификатор>]$_{опц}$[<тип_на_функция>] $_{опц}$ <име_на_функция>
(<формални_параметри>)
{<тяло>
}
където
<модификатор>::= inline|static| ...
<тип_на_функцията> ::= <име_на_тип> | <дефиниция_на_тип>
<име_на_функция> ::= <идентификатор>
<формални_параметри> :: <празно> | void |
<параметър> {, <параметър>}
<параметър> ::= <тип>[ & |$_{орс}$ * [const]$_{орс}$] $_{орс}$ <име_на_параметър>
<тип> ::= <име_на_тип>

```
<име_на_параметър> ::= <идентификатор>
<тяло> ::= <редица_от_оператори_и_дефиниции>
```

Фиг. 8.4 Дефиниция на функция

*Модификаторите* са спецификатори, които задават препоръка за компилатора (inline), класа памет (extern или static) и др. характеристики. Ще дадем примери в следващите разглеждания. Ако <модификатор> е пропуснат, подразбира се extern.

*Типът на функцията* е произволен тип без масив и функционален, но се допуска да е указател към такива обекти (в широкия смисъл на думата). Ако е пропуснат, подразбира се int.

*Името на функцията* е произволен идентификатор. Допуска се нееднозначност.

*Списъкът от формални параметри* (нарича се още **сигнатура**) може да е празен или void. Например, следната функция извежда текст:

```
void printtext(void)
{cout << "C++ Programming Language \n"
 cout << "B. Stroustrup \n";
 return;
}
```

В случай, че списъкът е непразен, имената на параметрите трябва да са различни. Те заедно с името определят еднозначно функцията. Формалните параметри са: *параметри – стойности*, *параметри – указатели* и *параметри – псевдоними*. Името на параметъра се предшества от тип.

*Примери:*

int a, int const& b, double& x, int const * y, const int* a

Засега няма да използваме параметри, специфицирани със const.

*Тялото на функцията* е редица от дефиниции и оператори. Тя описва алгоритъма, реализиращ функцията. Може да съдържа един или повече оператора return.

Операторът return (Фиг. 8.5) връща резултата на функцията в мястото на извикването.

**Оператор return**

*Синтаксис*

return [<израз>]$_{опц}$

където

- return е запазена дума;
- <израз> е произволен израз от тип <тип_на_функцията> или съвместим с него. Ако типът на функцията е void, <израз> се пропуска. В този случай е възможно и return да се пропусне.

*Семантика*

Пресмята се стойността на <израз>, конвертира се до типа на функцията (ако е възможно) и връщайки получената стойност в мястото на извикването на функцията, прекратява изпълнението ѝ.

Фиг. 8.5 Оператор return

*Забележка:* Ако функцията не е от тип void, тя задължително трябва да върне стойност. Това означава, че операторът return трябва да се намира във всички разклонения на тялото. В противен случай, повечето компилатори ще изведат съобщение или предупреждение за грешка. Възможно е обаче функцията да върне случайна стойност, което е лошо. По-добре е функцията да върне някаква безобидна стойност, отколкото случайна.

Функциите могат да се дефинират в произволно място на програмата, но не и в други функции. Преди да се извика една функция, тя трябва да е "позната" на компилатора. Това става, като дефиницията на функцията се постави пред main или когато функцията се дефинира на произволно място в частта за дефиниране на функции, а преди дефинициите на функциите се постави само нейната декларация (Фиг. 8.6).

**Декларация на функция**

<декларация_на_функция> ::=

[<модификатор>][<тип_на_резултата>]<име_на_функция>

([<формални_параметри>]);

Фиг. 8.6 Декларация на функция

Възможно е имената на параметрите във <формални_параметри> да се пропуснат.

**Семантика**

Описанието на функция задава параметрите, които носят входа и изхода, типа на резултата, а също и алгоритъма, за реализиране на действията, което функцията дефинира. Параметрите-стойности най-често задават входа на функцията. Параметрите-указатели и псевдоними са входно-изходните параметри за нея. Алгоритъмът се описва в тялото на функцията. Изпълнението на функцията завършва при достигане на края на тялото или след изпълнение на оператор return [<израз>]$_{опц}$;.

## 8.4 Обръщение към функция

*Синтаксис*

<обръщение_към_функция> ::=
<име_на_функция>()|
<име_на_функция>(void)|
<име_на_функция>(<фактически_параметри>)

където <фактически_параметри> са толкова на брой, колкото са формалните параметри. Освен по брой, формалните и фактическите параметри трябва да си съответстват по тип, по вид и по смисъл.

Съответствието по тип означава, че типът на i-тия фактически параметър трябва да съвпада (да е съвместим) с типа на i-тия формален параметър. Съответствието по вид се състои в следното: ако формалният параметър е параметър-указател, съответният му фактически параметър задължително е променлива или адрес на променлива, ако е параметър-псевдоним, съответният му фактически параметър задължително е променлива (за реализацията Visual C++, 6.0 от същия тип) и ако е параметър-стойност – съответният му фактически параметър е израз.

*Семантика*

Обръщението към функция е унарна операция с най-висок приоритет и с операнд - името на функцията. Последното пък е указател със стойност адреса на мястото в паметта където е записан програмният код

на функцията. Ако функцията определя процедура, обръщението към нея се оформя като оператор (завършва с ;). Опитът за използването й като израз предизвиква грешка. Ако функцията връща резултат както чрез return, така и чрез някой от формалните си параметри, обръщението към нея може да се разглежда и като оператор, и като израз. И ако функцията връща резултат единствено чрез оператора return, обръщението към нея има единствено смисъла на израз. Използването му като оператор не води до грешка, но не предизвиква видим резултат.

Обръщението към функция предизвиква генериране на нова стекова рамка и се осъществява на следните два етапа:

*1. Свързване на формалните с фактическите параметри*

За целта първият формален параметър се свързва с първия фактически, вторият формален параметър се свързва с втория фактически и т.н. последният формален параметър се свързва с последния фактически параметър. Свързването се реализира по различни начини в зависимост от вида на формалния параметър.

*а) формален параметър – стойност*

В този случай се намира стойността на съответния му фактически параметър. В стековата рамка на функцията за формалния параметър се отделя толкова памет, колкото типът му изисква и в нея се откопирва стойността на фактическия параметър.

*б) формален параметър – указател*

В този случай в стековата рамка на функцията за формалния параметър се отделят 4в, в които се записва стойността на фактическия параметър, която е адрес на променлива. Действията, описани в тялото се изпълняват със съдържанието на формалния параметър - указател. По такъв начин е възможна промяна на стойността на променливата, чийто адреа е предаден като фактически параметър.

*в) формален параметър – псевдоним*

Формалният параметър-псевдоним се свързва с адреса на фактическия. За него в стековата рамка на функцията памет не се отделя. Той просто “прелита” и се “закачва” за фактическия си параметър. Действията с него се извършват над фактическия параметър.

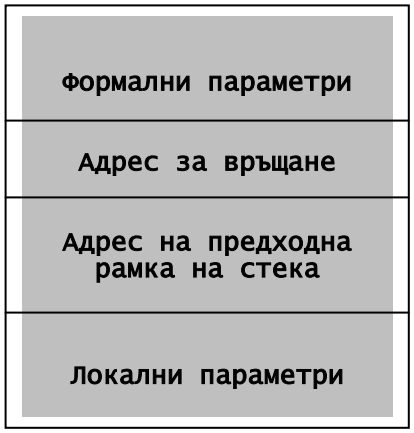
*2. Изпълнение на тялото на функцията*

Аналогично е на изпълнението на блок.

При всяко обръщение към функция в програмния стек се включва нов “блок” от данни. В него се съхраняват формалните параметри на функцията, нейните локални променливи, а също и някои “вътрешни” данни като return-адреса и др. Този блок се нарича **стекова рамка на функцията**.

В дъното на стека е стековата рамка на main. На върха на стека е стековата рамка на функцията, която се обработва в момента. Под нея е стековата рамка на функцията, извикала функцията, обработваща се в момента. Ако изпълнението на една функция завършва, нейната стекова рамка се отстранява от стека.

Видът на стековата рамка зависи от реализацията. С точност до наредба, тя има вида:




8.7 Стекова рамка

**Област на идентификаторите в програмата на C++**

Идентификаторите означават имена на константи, променливи, формални параметри, функции, класове. Най-общо казано, има три вида области на идентификаторите: *глобална*, *локална* и *област за клас*. Областите се задават *неявно* – чрез позицията на идентификатора в програмата и *явно* – чрез декларация. Отново разглеждането ще е непълно, заради пропускането на класовете и явното задаване на област.

*Глобални идентификатори*

Дефинираните пред всички функции константи и променливи могат да се използват във всички функции на модула, освен ако не е дефиниран локален идентификатор със същото име в някоя функция на модула. Наричат се **глобални идентификатори**, а областта им – **глобална.**

*Локални идентификатори*

Повечето константи и променливи имат локална област. Те са дефинирани вътре във функциите и не са достъпни за кода в другите функции на модула. Областта им се определя според общото правило – започва от мястото на дефинирането и завършва в края на оператора (блока), в който идентификаторът е дефиниран. Формалните параметри на функциите също имат локална видимост. Областта им е тялото на функцията.

В различните области могат да се използват еднакви идентификатори. Ако областта на един идентификатор се съдържа в областта на друг, последният се нарича нелокален за първоначалния. В този случай е в сила правилото: Локалният идентификатор “скрива” нелокалния в областта си.

Областта на функция започва от нейното дефиниране и продължава до края на модула, в който функцията е дефинирана. Ако дефинициите на функциите са предшествани от тяхните декларации, редът на дефиниране на функциите в модула не е от значение – функциите са видими в целия модул. Препоръчва се също дефинирането на заглавен файл с прототипите (декларациите) на използваните функции.

## 8.5 Масивите като формални параметри

**Едномерни масиви**

Съществуват различни начини за задаване на формални параметри от тип едномерен масив.

*а) традиционен*

Дефиницията

```
T a[]
```

където T е скаларен тип, задава параметър a от тип едномерен масив с базов тип T. Може да се укаже горна граница на масива, но компилаторът я пренебрегва.

*Примери:*

int a[]  - a е параметър от тип масив от цели числа,

int a[10]  - еквивалентна е на int a[],

double b[]  - b е параметър от тип масив от реални числа,

char c[]  - c е параметър от тип масив от символи.

*б) чрез указател*

Дефиницията

T* p

където T е скаларен тип, задава параметър p от тип указател към тип T. От връзката между масив и указател следва, че тази дефиниция може да се използва и за дефиниране на формален параметър от тип масив.

*Примери:* Следните дефиниции на формални параметри са еквивалентни на тези от примера по-горе:

int* a  - a е параметър от тип указател към int

double* b  - b е параметър от тип указател към double.

char* c  - c е параметър от тип указател към char.

И в двата случая фактическият параметър се указва с името на едномерен масив от същия тип. Необходимо е също на функцията да се подаде като параметър и размерът на масива.

**Задача 70.** Да се напишат функции, които въвеждат и извеждат елементите на едномерен масив от цели числа. Като се използват тези функции да се напише програма, която въвежда редица от естествени числа, след което я извежда, а също извежда най-големия общ делител на елементите на редицата.

Програма Zad70.cpp решава задачата.

```
// Program Zad70.cpp
#include <iostream.h>
int gcd(int, int);
void readarr(int, int[]);
void writearr(int, int[]);
int main()
{cout << "n= ";
 int n;
 cin >> n;
```

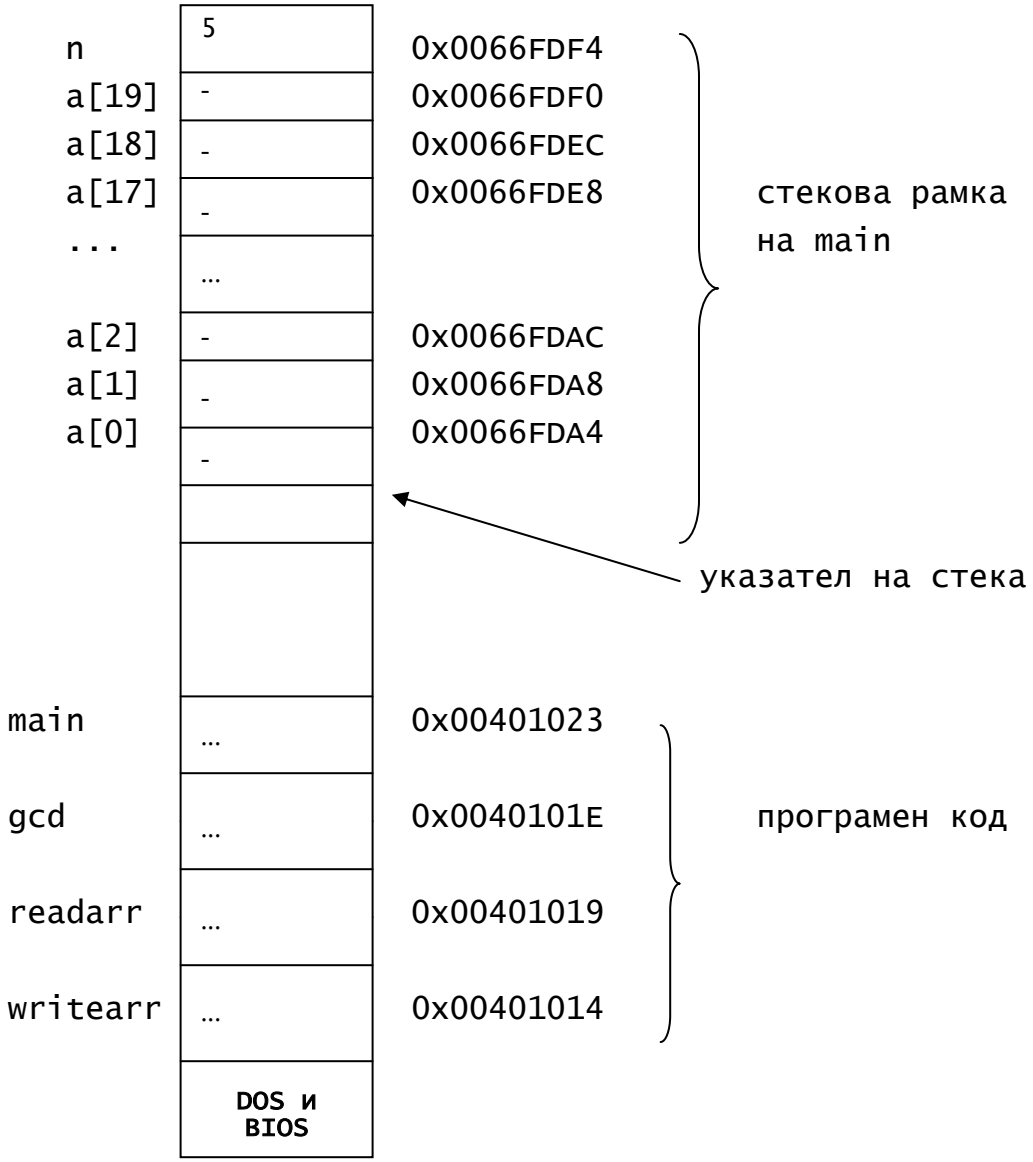
```
 int a[20];
 readarr(n, a);
 writearr(n, a);
 int x = a[0];
 for (int i = 1; i <= n-1; i++)
  x = gcd(x, a[i]);
 cout << "gcd = " << x << '\n';
 return 0;
}
int gcd(int a, int b)
{while (a != b)
 if (a > b) a = a-b; else b = b-a;
 return a;
}
void readarr(int m, int arr[])
// m е размерността на масива
// arr е едномерен масив
{for (int i = 0; i <= m-1; i++)
 {cout << "arr[" << i << "]= ";
  cin >> arr[i];
 }
}
void writearr(int m, int arr[])
// m е размерността на масива
// arr е едномерен масив
{for (int i = 0; i <= m-1; i++)
 cout << "arr[" << i << "]= " << arr[i] << ‘\n’;
}
```

*Изпълнение на програмата*

Фрагментът

```
cout << "n= ";
int n;
cin >> n;
int a[20];
```

дефинира и въвежда стойност на n, а също дефинира променлива a от тип масив. Нека за n е въведено 5. В резултат е създадена стековата рамка на main. ОП до този момент има вида:




Обръщението readarr(n, a); се реализира като се свързват формалните с фактическите параметри и се изпълни тялото. За целта се формира нова стекова рамка – тази на readarr, в която за формалния параметър arr се отделят 4B, в която памет се откопирва стойността на фактическия параметър a (адресът на a[0]), за m се отделят също 4B, в които се откопирва 5 – стойността на фактическия параметър n. Тялото на функцията се изпълнява като блок. Операторът за цикъл

```
for (int i = 0; i <= m-1; i++)
```

```
     {cout << "arr[" << i << "]= ";
      cin >> arr[i];
      }
```

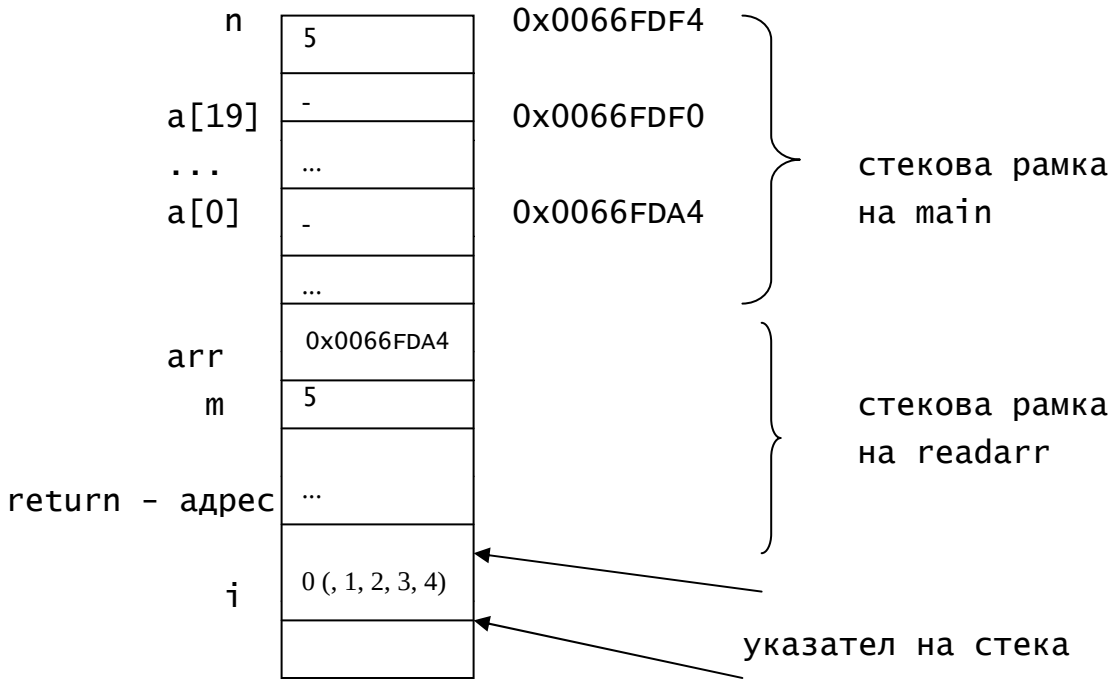
е еквивалентен на

```
     for (int i = 0; i <= m-1; i++)
     {cout << "arr[" << i << "]= ";
      cin >> *(arr + i);
      }
```

и се изпълнява по следния начин: За цялата променлива i се отделят 4в в стековата рамка на readarr. i последователно приема стойностите 0, 1, ..., 4 и за всяка стойност се изпълнява блокът

```
     {cout << "arr[" << i << "]= ";
      cin >> *(arr + i);
      }
```

Операторът cin >> *(arr + i); въвежда стойност на индексираната променлива a[i], тъй като arr + i е адреса на i-тия елемент на a, а *(arr+i) е неговата стойност. Така във функцията се работи с формалния параметър arr, а в действителност действията се изпълняват с фактическия параметър – едномерния масив a. Функцията readarr работи с масива a, а не с негово копие.



След достигане на края на функцията изпълнението й завършва и се освобождава стековата й рамка. В резултат, първите 5 елемента на масива a получават текущи стойности. Операторът

```
writearr(n, a);
```

се изпълнява по аналогичен начин. Отново се работи с фактическия параметър - масива a, а не с негово копие. В този случай, елементите a[0], a[1], ..., a[n-1] на a само се сканират и извеждат. Не се извършват промени над тях. За да ги защитим от неправомерен достъп, е добре формалният параметър arr да дефинираме като указател към цяла константа, т.е. като const int arr[]. Тогава всеки опит да се променя arr[i] (i = 0, 1, ..., n-1) в writearr ще предизвика грешка.

Фрагментът

```
int x = a[0];
for (int i = 1; i <= n-1; i++)
   x = gcd(x, a[i]);
```

намира най-големия общ делител на елементите на редицата.

В заглавията на последните две процедури горните граници на индексите могат явно да се укажат. Например

```
void writearr(int m, int arr[20])
```

и

```
void readarr(int m, int arr[20])
```

са валидни заглавия, но компилаторът не се нуждае от горната граница. Трябват му само скобите [], за да разпознае параметър от тип масив. Може също да се използва второто представяне на формален параметър от тип масив, т.е.

```
void writearr(int m, int* arr)
```

и

```
void readarr(int m, int* arr)
```

Тези представяния на формалните параметри са напълно еквивлентни.

**Забележки:**

1. Функциите readarr и writearr работят с направо с масива a, а не с негови копия. Промените на елементите на масива се запазват след излизане от функцията.
2. Размерът на масивът не може да се разбере от неговото описание. Затова се налага използването на допълнителния параметър m в списъка от аргументи на функциите. Последното не

се отнася за масивите, представляващи символни низове, тъй като те завършват със знака за край на низ ‘\0’.

**Задача 71.** Да се напише функция len(char* s), която намира дължината на символен низ, а също функция eqstrs(char*, char*), която сравнява два символни низа за лексикографско равно.

Функциите Zad71_1 и Zad71_2 решават задачата.

```
// Function Zad71_1
int len(char* s)
{int k = 0;
 while(*s)
 {k++;
  s++;
 }
 return k;
}

// Function Zad71_2
bool eqstrs(char* str1, char* str2)
{while (*str1 == *str2 && * str1)
  {str1++; str2++;}
 return *str1 == *str2;
 }
```

Обърнете внимание, че тъй като всеки низ завършва със символа ‘\0’, който се интерпретира като false, изразът *s в оператора за цикъл while на функцията len, ще бъде истина и тялото ще се изпълнява до достигане на края на низа. Аналогична конструкция имаме и при дефиницията на функцията eqstrs.

**Задача 72.** Да се напише булева функция, която проверява дали цялото число x е елемент на редицата от цели числа $a_0$, $a_1$, ..., $a_{n-1}$.

Функцията Zad72 решава задачата.

```
// Function Zad72
```

```
bool search(int n, int a[], int x)
{int i = 0;
 while (a[i] != x && i < n-1)i++;
 return a[i]==x;
}
```

**Допълнение:** Обръщението

search(m, b, y)

проверява дали елементът y се съдържа в редицата $b_0$, $b_1$, ..., $b_{m-1}$, а

search(k, b + m, y)

проверява дали y се съдържа в подредицата $b_m$, $b_{m+1}$, ..., $b_{m+k-1}$.

Еквивалентна дефиниция на тази функция е:

```
bool search(int n, int* a, int x)
{int i = 0;
 while (*(a+i) != x && i < n-1)i++;
 return *(a+i)==x;
}
```

**Задача 73.** Да се напише функция, която проверява дали редицата от цели числа $a_0$, $a_1$, ..., $a_{n-1}$ е монотонно намаляваща.

Функцията Zad73 решава задачата.

```
// Function Zad73
bool monnam(int n, int a[])
{int i = 0;
 while (a[i] >= a[i+1] && i < n-2)i++;
 return a[i] >= a[i+1];
}
```

или

```
bool monnam(int n, int* a)
{int i = 0;
 while (*(a+i) >= *(a+i+1) && i < n-2)i++;
 return *(a+i) >= *(a+i+1);
}
```

**Допълнение:** Обръщението

monnam(m, b)

проверява дали редицата $b_0$, $b_1$, ..., $b_{m-1}$ е монотонно намаляваща, а
monnam(k, b + m)
- дали подредицата $b_m$, $b_{m+1}$, ..., $b_{m+k-1}$ на $b_0$, $b_1$, ..., $b_{m-1}$ е монотонно намаляваща.

**Задача 74.** Да се напише функция, която проверява дали редицата от цели числа $a_0$, $a_1$, ..., $a_{n-1}$ се състои от различни елементи.

Функцията Zad74 решава задачата.

```
// Function Zad74
bool differ(int n, int a[]
)
{int i = -1;
 bool b; int j;
 do
 {i++; j = i;
  do
  {j++;
   b = a[i] != a[j];
  }while (b && j < n-1);
  }while (b && i < n-2);
  return b;
 }
```

**Допълнение:** Обръщението
differ(m, b, y)
проверява дали редицата $b_0$, $b_1$, ..., $b_{m-1}$ се състои от различни елементи, а
differ(k, b + m)
- дали подредицата $b_m$, $b_{m+1}$, ..., $b_{m+k-1}$ на $b_0$, $b_1$, ..., $b_{m-1}$ се състои от различни елементи.

**Задача 75.** Да се напише програма, която въвежда две числови редици, сортира ги във възходящ ред, слива ги и извежда получената редица.

Програма Zad75.cpp решава задачата. За целта са дефинирани следните функции:

readarr – въвежда числова редица
writearr – извежда числова редица върху екрана
sortarr - сортира във възходящ ред елементите на числова редица
mergearrs - слива числови редици.

```
// Program Zad75.cpp
#include <iostream.h>
#include <iomanip.h>
void writearr(int, double[]);
void readarr(int, double[]);
void sortarr(int, double[]);
void mergearrs(int, double[], int, double[], int&, double[]);

int main()
{cout << "n= ";
 int n;
 cin >> n;
 double a[20];
 readarr(n, a);
 cout << endl;
 writearr(n, a);
 cout << endl;
 sortarr(n, a);
 cout << endl;
 writearr(n, a);
 cout << "m= ";
 int m;
 cin >> m;
 double b[30];
 readarr(m, b);
 cout << endl;
 writearr(m, b);
 cout << endl;
 sortarr(m, b);
```

```
 cout << endl;
 writearr(m, b);
 cout << endl;
 int p;
 double c[50];
 mergearrs(n, a, m, b, p, c);
 writearr(p, c);
 return 0;
}
void writearr(int m, double arr[])
{cout << setprecision(3) << setiosflags(ios::fixed);
 for (int i = 0; i <= m-1; i++)
   cout << setw(10) << arr[i];
 cout << "\n";
}
 void readarr(int m, double arr[])
 {for (int i = 0; i <= m-1; i++)
 {cout << "arr[" << i << "]= ";
  cin >> arr[i];
 }
 }
  void sortarr(int n, double a[])
 {for (int i = 0; i <= n-2; i++)
 {int k = i;
  double min = a[i];
  for (int j = i+1; j <= n-1; j++)
    if (a[j] < min)
    {min = a[j];
     k = j;
    }
  double x = a[i]; a[i] = a[k]; a[k] = x;
 }
}
void mergearrs(int n, double a[], int m, double b[],
               int& k, double c[])
{int i = 0, j = 0;
```

```
k = -1;
while (i <= n-1 && j <= m-1)
if (a[i] <= b[j])
{k++;
 c[k] = a[i];
 i++;
}
else
{k++;
 c[k] = b[j];
 j++;
}
int l;
if (i > n-1)
  for (l = j; l <= m-1; l++)
  {k++;
   c[k] = b[l];
  }
else
  for (l = i; l <= n-1; l++)
  {k++;
   c[k] = a[l];
  }
  k++;
}
```

**Многомерни масиви**

Когато многомерен масив трябва да е формален параметър на функция, в описанието му трябва да присъстват като константи всички размери с изключение на първият. Например, декларацията

```
void readarr2(int n, int matr[][20]);
```

определя matr като двумерен масив (редица от двадесеторки от цели числа). Описанието

```
int (*matr)[20]
```

е еквивалентно на

```
int matr[][20]
```

Скобите, ограждащи *matr, са задължителни. В противен случай, тъй като [] е с по-висок приоритет от *, int *matr[20] ще се интерпретира като “matr е масив с 20 елемента от тип *int”.

**Задача 76.** Да се напише програма, която въвежда квадратна матрица от цели числа, след което я извежда като увеличава всеки от елементите на матрицата над главния диагонал с 5 и намалява всеки от елементите под главния диагонал с 5.

Програма Zad76.cpp решава задачата. Тя дефинира функциите:
readarr2 – въвежда квадратна матрица
writearr2 – извежда квадратна матрица
transff - увеличава всеки от елементите на матрицата над главния диагонал с 5 и намалява всеки от елементите под главния диагонал с 5.

```
// Program Zad76.cpp
#include <iostream.h>
#include <iomanip.h>
void readarr2(int, int[][10]);
void writearr2(int, int[][10]);
void transff(int, int[][10]);
int main()
{int a[10][10];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error. Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 10)
 {cout << "Incorrect input! \n";
  return 1;
 }
 readarr2(n, a);
 cout << '\n';
```

```
 writearr2(n, a);
 cout << '\n';
 transff(n, a);
 writearr2(n, a);
 return 0;
}
void readarr2(int n, int arr[][10])
{for (int i = 0; i <= n-1; i++)
  for (int j = 0; j <= n-1; j++)
   cin >> arr[i][j];
}
void writearr2(int n, int arr[][10])
{for (int i = 0; i <= n-1; i++)
 {for (int j = 0; j <= n-1; j++)
   cout << setw(5) << arr[i][j];
  cout << "\n";
 }
}
void transff(int n, int arr[][10])
{int i, j;
 for (i = 1; i <= n-1; i++)
  for (j = 0; j <= i-1; j++)
     arr[i][j] = arr[i][j] - 5;
 for(i = 0; i <= n-2; i++)
  for(j = i+1; j <= n-1; j++)
     arr[i][j] = arr[i][j] + 5;
}
```

Обръщението

```
transff(k, a + m);
```

ще извърши същото действие над квадратната подматрица на дадената матрица:

$$\begin{pmatrix} a_{m,0} & a_{m,1} & \dots & a_{m,\,k-1} \\ a_{m+1,\,0} & a_{m+1,1} & \dots & a_{m+1,k-1} \\ \dots & \dots & \dots & \dots \\ a_{m+k-1,0} & a_{m+k-1,1} & \dots & a_{m+k-1,k-1} \end{pmatrix}$$

**Задача 77.** Да се напише програма, която въвежда редица от думи не по-дълги от 14 знака и дума, също не по-дълга от 14 знака. Програмата да проверява дали думата се среща в редицата. За целта да се оформят подходящи функции.

Програма Zad77.cpp решава задачата. В нея са дефинирани функциите:

void readarrstr(int n, char s[][15]) - въвежда редица от n думи,

bool search(int n, char s[][15], char* x) - търси думата x в редицата s от n думи. За целта използва помощната функция

bool eqstrs(char* str1, char* str2);

от Задача 71.

```
// Program Zad77.cpp
#include <iostream.h>
#include <string.h>
void readarrstr(int, char [][15]);
bool eqstrs(char*, char*);
bool search(int, char [][15], char*);
int main()
{char a[20][15];
 cout << "n= ";
 int n;
 cin >> n;
 readarrstr(n, a);
 cout << "word: ";
 char word[15];
 cin >> word;
 if (search(n, a, word)) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
void readarrstr(int n, char s[][15])
{for(int i = 0; i <= n-1; i++)
 {cout << "s[" << i << "]= ";
  cin >> s[i];
```

```
 }
}
bool eqstrs(char* str1, char* str2)
 {while (*str1 && *str1 == *str2)
   {str1++;
    str2++;
   }
  if(*str1 != *str2) return false;
  else return true;
 }
bool search(int n, char s[][15], char* x)
{int i = 0;
 while (!eqstrs(s[i], x) && i < n-1) i++;
 return eqstrs(s[i], x);
}
```

**Задача 78.** Да се напише програма, която умножава две матрици.

Програма Zad78.cpp решава задачата. Тя дефинира следните функции:
readarr2 – въвежда матрица,
writearr2 – извежда матрица,
multmatr – умножава матрици.

```
// Program Zad78.cpp
#include <iostream.h>
#include <iomanip.h>
void readarr2(int n, int m, double [][30]);
void writearr2(int n, int m, double [][30]);
void multmatr(int, int, int, double [][30],
           double [][30], double [][30]);

int main()
{double a[10][30], b[20][30], c[10][30];
 cout << "n= ";
 int n;
 cin >> n;
```

```
if (!cin)
{cout << "Error. Bad input! \n";
 return 1;
}
if (n < 1 || n > 10)
{cout << "Incorrect input! \n";
 return 1;
}
cout << "m= ";
int m;
cin >> m;
if (!cin)
{cout << "Error. Bad input! \n";
 return 1;
}
if (m < 1 || m > 30)
{cout << "Incorrect input! \n";
 return 1;
}
 cout << "k= ";
int k;
cin >> k;
if (!cin)
{cout << "Error. Bad input! \n";
 return 1;
}
if (k < 1 || k > 30)
{cout << "Incorrect input! \n";
 return 1;
}
readarr2(n, m, a);
writearr2(n, m, a);
cout << "\n";
readarr2(m, k, b);
cout << "\n";
writearr2(m, k, b);
```

```
 cout << "\n";
 multmatr(n, m, k, a, b, c);
 writearr2(n, k, c);
 return 0;
}
void readarr2(int n, int m, double arr [][30])
{for (int i = 0; i <= n-1; i++)
   for (int j = 0; j <= m-1; j++)
      cin >> arr[i][j];
 return;
}
void writearr2(int n, int m, double arr[][30])
{cout << setprecision(3) << setiosflags(ios::fixed);
 for (int i = 0; i <= n-1; i++)
 {for (int j = 0; j <= m-1; j++)
   cout << setw(10) << arr[i][j];
  cout << "\n";
 }
 return;
}
void multmatr(int n, int m, int k, double a[][30],
              double b[][30], double c[][30])
{for (int i = 0; i <= n-1; i++)
  for (int j = 0; j <= m-1; j++)
  {c[i][j] = 0;
   for (int p = 0; p <= m-1; p++)
      c[i][j] += a[i][p] * b[p][j];
  }
}
```

## 8.6 Масивите като върнати оценки

Въпреки, че масивите могат да са параметри на функции, функциите **не могат** да са от тип масив. Възможно е обаче да са от тип указател. Това позвлява дефинирането на функции, които връщат масиви.

*Пример:* В следващата програма е дефинирана функцията readarr, която въвежда стойности на едномерен масив. Тя връща резултат не само чрез променливата от тип масив arr, но и чрез оператора return. Това позволява обръщенията към нея да служат както за оператори, така и за изрази.

```
#include <iostream.h>
void writearr(int, int[]);
int* readarr(int, int[]);
int main()
{cout << "n= ";
 int n;
 cin >> n;
 int a[20];
 int* p = readarr(n, a);
 writearr(n, p);
 cout << endl;
 return 0;
}
void writearr(int m, int arr[])
{for (int i = 0; i <= m-1; i++)
 cout << "arr[" << i << "]= " << arr[i] << '\n';
return;
}
int* readarr(int m, int arr[])
{for (int i = 0; i <= m-1; i++)
 {cout << "arr[" << i << "]= ";
  cin >> arr[i];
 }
return arr;
}
```

**Въпрос:** Допустима ли е конструкцията: readarr(n, a)[i], където i е цяло число от 0 до n-1? Ако това е така, какъв е резултатът от изпълнението му?

**Задача 79.** Да се напише функция, която намира и връща като резултат конкатенацията на два низа. Функцията да променя първия си

аргумент като в резултат той също да съдържа конкатенацията на низовете.

Програма Zad79.cpp решава задачата.

```
// Program Zad79.cpp
#include <iostream.h>
int len(char*);
char *cat(char*, char*);
int main()
{char s1[100];
 cout << "s1= ";
 cin >> s1;
 cout << "s2= ";
 char s2[100];
 cin >> s2;
 cout << cat(s1, s2) << "  " << s1 << '\n';
 return 0;
}
int len(char* s)
{int k = 0;
 while (*s)
 {k++; s++;
 }
 return k;
}
char* cat(char *s1, char *s2)
{int i = len(s1);
 while (*s2)
 {s1[i] = *s2;
  i++;
  s2++;
}
 s1[i] = '\0';
 return s1;
}
```

## Задачи

**Задача 1.** Въпреки многото й недостатъци, следващата програма е доста поучителна. Тя дефинирана функцията readarr, която има за формален параметър броя на елементите на масива и връща едномерен масив, определен чрез указател към първия му елемент.

```
#include <iostream.h>
int a[20];
void writearr(int, int[]);
int* readarr(int);
int main()
{cout << "n= ";
 int n;
 cin >> n;
 int* p = readarr(n);
 writearr(n, p);
 cout << '\n';
 return 0;
}
void writearr(int m, int arr[])
{for (int i = 0; i <= m-1; i++)
 cout << "arr[" << i << "]= " << arr[i] << '\n';
}
int* readarr(int m)
{for (int i = 0; i <= m-1; i++)
{cout << "a[" << i << "]= ";
 cin >> a[i];
}
return a;
}
```

Извършете експерименти с тази програма.

**Задача 2.** Да се напише програма, която въвежда полиномите:

$$P_n(x) = a_o x^{n-1} + a_1 x^{n-2} + \ldots + a_{n-2} x^1 + a_{n-1},$$

$$R_k(x) = r_o x^{k-1} + r_1 x^{k-2} + \ldots + r_{k-2} x^1 + r_{k-1}$$

$$Q_n(x) = b_o x^{n-1} + b_1 x^{n-2} + \ldots + b_{n-2} x^1 + b_{n-1},$$

и реалната променлива x и намира и извежда стойностите на полиномите в x.

**Задача 3.** Да се напише програма, която въвежда стойности на редиците:

$a_0, a_1, \ldots, a_{n-1},$

$b_0, b_1, \ldots, b_{m-1},$

$c_0, c_1, \ldots, c_{p-1}$

и намира и извежда AR1, AR2, BR1, BR2, CR1 и CR2, където за дадена редица $x_0, x_1, \ldots, x_{k-1}$

$$XR1 = \frac{1}{k}\sum_{i=0}^{k-1} x_i, \qquad XR2 = \frac{1}{k}\sqrt{\sum_{i=0}^{k-1}(x_i - XR1)^2}.$$

**Задача 4.** Да се напише функция, която намира разстоянието между две точки в равнината, зададени чрез координатите си (x1, y1) и (x2, y2). Като се използва тази функция да се напише програма, която чете координатите на n точки ($n \geq 1$) от равнината и намира и извежда разстоянието между всеки две от тях.

**Задача 5.** да се напише функция, която връща стойност true, ако a, b и c са страни на триъгълник и false - в противен случай. Като се използва тази функция, да се напише програма, която въвежда стойности на елементите на матрицата $A_{3xn}$ и определя кои от тройките (a[0][i], a[1][i], a[2][i]), i = 0, 1, ..., n-1 могат да служат за страни на триъгълник.

**Задача 6.** Да се напише функция, която връща стойност true ако редицата от цели числа $x_0, x_1, \ldots, x_{k-1}$ има поне два последователни нулеви елемента. Като се използва тази функция, да се напише програма, която намира и извежда номерата на редовете на матрицата A [n x n], от цели числа, които имат поне два последователни нулеви елемента.

**Задача 7.** Да се напише функция, която намира сумата на два полинома. Като се използва тази функция, да се напише програма, която намира сумата на всеки два от полиномите:

$$P_n(x) = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x^1 + a_0,$$

$$Q_m(x) = b_{n-1} x^{m-1} + b_{n-2} x^{m-2} + \ldots + b_1 x^1 + b_0,$$

$$R_k(x) = r_{k-1} x^{k-1} + r_{k-2} x^{k-2} + \ldots + r_1 x^1 + r_0,$$

**Задача 8.** Даден е триъгълник със страни a, b и c. Да се напише програма, която намира медианите на триъгълник, страните на който са медианите на дадения триъгълник.

*Упътване:* Медианата към страната a на триъгълника е равна на

$$\frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}.$$

**Задача 9.** Дадени са координатите на върховете на n триъгълника. Да се напише програма, която определя, кой от триъгълниците е с по-голямо лице.

**Задача 10.** Дадени са естественото число p > 1 и реалните квадратни матрици с размерности n x n - A, B и C. Да се напише програма, която намира матрицата

$$(A \,.\, B \,.\, C)^p.$$

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
3. Д. Луис, C/C++ бърз справочник, ИнфоДАР, София, 1998.
4. И. Момчев, К. Чакъров, Програмиране III, C и C++, ТУ, София, 1996.
5. М. Тодорова, Програмиране на Паскал, Полипринт, София, 1993.

# 9

# Функции от по-висок ред

Функция, някои формални параметри на която са функции, се нарича **функция от по-висок ред.**

В езика C++ е възможно формален параметър на функция да е указател към функция, а също е възможно резултатът от изпълнението на функция да е указател към функция. Това позволява да се реализират функции от по-висок ред, а също и такива, които връщат функция.

## 9.1 Указател към функция

Името на функция е константен указател, сочещ към първата машинна инструкция от изпълнимия й машинен код. В езика C++ е възможно да се дефинират променливи, които са указатели към функции (Фиг. 9.1).

**Дефиниция на указател към функция**

<дефиниция_на_променлива_указател_към_функция> ::=

<тип_на_функция>(*<указател_към_функция>)(<формални_параметри>)

[= <име_на_функция>]$_{опц}$;

където

- <указател_към_функция> е идентификатор;
- <име_на_функция> е идентификатор, означаващ име на функция от тип <тип_на_функция> и параметри - <формални_параметри>;
- <тип_на_функция> и <формални_параметри> се дефинират аналогично на съответните от заглавието на дефиниция функция. Имената на параметрите могат да се пропуснат.

Фиг. 9.1 Дефиниция на указател към функция

**Забележка:** Скобите, ограждащи *<указател_към_функция>, са задължителни. В противен случай дефиницията ще се изтълкува от компилатора като декларация на функция с име < указател_към_функция>, с параметри – <формални_параметри> и тип – указател към <тип_на_функция>.

В резултат на дефиницията на променлива от тип указател към функция, за променливата се отделят 4В ОП, която е с неопределена стойност, ако дефиницията е без инициализация, и съдържа адреса на първата машинна команда от изпълнимия код на функцията, чрез която е направена инициализацията, ако дефиницията е с инициализация.

*Примери:*

1. double (*p)(double, double);

е дефиниция на променлива p от тип указател към функция от тип double с два аргумента също от тип double. В резултат за p се отделят 4В ОП, които са с неопределена стойност.

2. int (*q)(int, int*);

дефинира променлива q от тип указател към функция от тип int, с два аргумента, единият от които цял, а другият – указател към int. За q се отделят 4В ОП, които са с неопределена стойност.

Нека са дефинирани следните функции за сортиране на числови редици:

```
void bubblesort(int, int*);   // метод на мехурчето
void mergesort(int, int*);    // сортиране чрез сливане
void heapsort(int, int*);   // пирамидално сортиране
```

Променливата r може да е указател към тези функции ако е дефинирана по следния начин:

```
void (*r)(int, int*);
```

r не може да е указател към функциите:

```
int f1(int, int*);
int f2(int, int*);
```

Указател към последните може да е променливата s, където:

```
int (*s)(int, int*);
```

Горните дефиниции на p, q, r и s са без инициализации.

Дефиниците на променливите x и y

```
void (*x)(int, int*) = bubbesort;
```

```
int (*y)(int, int*) = f2;
```

са с инициализация. За всяка от тях се отделят 4В ОП, в която памет се записват адресите на първите команди на изпълнимите кодове на bubblesort и f2 съответно.

На променлива от тип указател към функция може да се присвои името на функция от същия тип. Присвояването се извършва по общоприетия начин.

*Пример:* Допустими са присвояванията:

```
r = mergesort;
x = heapsort;
```

Обръщението към функция освен директно може да се осъществява и индиректно – чрез указател към нея. След инициализация на променлива от тип указател към функция, чрез променливата може да се осъществи обръщение към конкретна функция. Така се предоставя ефективен способ за предаване на управлението към потребителски и библиотечни функции.

*Пример:*

```
void (*r)(int, int*) = bubblesort;
bubblesort(n, a);     // директно обръщение
(*r)(n, a);           // индиректно обръщение (чрез r).
```

**Забележка:** Някой компилатори, в това число и на Visual C++ 6.0, допускат извикването на функция чрез указател да се осъществява и само чрез името на указателя.

*Пример:* Ако

```
void (*r)(int, int*) = bubblesort;
```

Обръщението към bubblesort

```
(*r)(n, a);
```

може да се запише и по следния начин: `r(n, a);`

## 9.2 Функциите като формални параметри

Указател към функция може да е формален параметър на функция. Ще илюстрираме тази възможност с няколко примери.

**Задача 80.** Да се напише функция, която реализира математическата абстракция:

$$\sum_{\substack{i=a \\ i\to next(i)}}^{b} f(i).$$

където a и b са дадени реални числа (a ≤ b), f е реална едноаргументна функция, задаваща терма, а next е реална едноаргументна функция, задаваща стъпката за промяна на управляващия параметър на сумата.

Преди да решим задачата в общияй вид ще предложим няколко частни решения.

а) Да се дефинира функция, която намира стойността на сумата:

sin(a) + sin(a+1) + sin(a+2) + ... + sin(b),

където a и b са дадени реални числа.

Функцията sum_sin намира тази сума.

```
double sum_sin(double a, double b)
{double s = 0;
 for (double i = a; i <= b + 1E-14; i = i + 1)
    s = s + sin(i);
 return s;
}
```

б) Да се дефинира функция, която намира стойността на сумата:

cos(a) + cos(a + 0.2) + cos(a + 0.4) + ... + sin(b)

където a и b са дадени реални числа.

Функцията sum_cos намира тази сума.

```
double sum_cos(double a, double b)
{double s = 0;
 for (double i = a; i <= b + 1e-14; i = i + 0.2)
     s = s + cos(i);
 return s;
}
```

Забелязваме, че тези две функции се “приличат”. Написани са по следния общ шаблон:

```
double <name>(double a, double b)
{double s = 0;
 for (double i = a; i <= b + 1e-14; i = <next>(i))
    s = s + <f>(i);
 return s;
}
```

Елементите, по които функциите sum_sin и sum_cos се различават са означени с <...> в шаблона. Това са две функции: f, означаваща терма и next - стъпката на сумата. Като използваме възможността формален параметър на функция да е указател към функция, можем да изнесем <f> и <next> като формални параметри на функцията и да обобщим тези частни случаи. Така стигаме до функцията sum:

```
// Function Zad80
double sum(double a, double b, double (*f)(double),
           double (*next)(double))
{double s = 0;
 for (double i = a; i <= b + 1e-14; i = next(i))
    s = s + f(i);
 return s;
}
```

Обръщенията към sum:

```
sum(a, b, sin, next1)
sum(a, b, cos, next2)
```

където

```
int next1(double x)
{return x + 1;
}
int next2(double x)
{return x + 0.2;
}
```

реализират горните частни случаи.

sum е функция от по-висок ред. В нея третият и четвъртият параметри са указатели към функции.

Като използваме sum, може да дефинираме sum_sin и sum_cos по следния начин:

```
double sum_sin(double a, double b)
{return sum(a, b, sin, next1);
}
double sum_cos(double a, double b)
{return sum(a, b, cos, next2);
}
```

**Задача 81.** Да се напише функция, която реализира математическата абстракция:

$$\prod_{\substack{i=a \\ i\to next(i)}}^{b} f(i)$$

където a и b са реални числа, f е реална едноаргументна функция, задаваща терма, а next - реална едноаргументна функция, задаваща стъпката за промяна на управляващия параметър на произведението. Да се включи тази функция в програма и се намерят:

tg(1) * tg(1.5) * tg(2) * tg(2.5) * tg(3)

и

arctg(1) * arctg(1.1) * arctg(1.2) * arctg(1.3).

Програма Zad81.cpp решава задачата. Функцията prod от нея се реализира чрез последователно преминаване през стъпки, аналигични на тези от задача 80.

```
// Program Zad81.cpp
#include <iostream.h>
#include <math.h>
double prod(double, double, double (*)(double),
            double (*) (double));
double next1(double);
double next2(double);
int main()
{cout << prod(1, 3, tan, next1) << '\n';
 cout << prod(1, 1.3, atan, next2) << '\n';
 return 0;
}
double prod(double a, double b, double (*f)(double),
            double (*next)(double))
{double s = 1.0;
 for (double i = a; i <= b + 1e-14; i = next(i))
    s = s * f(i);
 return s;
}
double next1(double x)
```

```
{return x + 0.5;
}
double next2(double x)
{return x + 0.1;
}
```

В тази програма е дефинирана функцията от по-висок ред prod, реализираща исканата абстракция. В нея третият и четвъртият параметри са указатели към функции. В главната програма са направени две обръщения към нея

```
prod(1, 3, tan, next1)
```

и

```
prod(1, 1.3, atan, next2),
```

намиращи търсените произведения.

Забелязваме, че функциите sum и prod си “приличат”. Написани са по следния общ шаблон.

```
double <name>(double a, double b, double (*f)(double),
              double (*next)(double))
{double s = <null_val>;
 for (double i = a; i <= b + 1e-14; i = next(i))
    s = s <op> f(i);
 return s;
}
```

И в този случай, елементите, по които sum и prod се различават са оградени с <...>. Това са операцията op и нулата на операцията - null_val. Отново ще изнесем op и null_val като формални параметри на функцията. Тъй като op е бинарна инфиксна операция, а не име на функция, ще дефинираме помощна реална функция с име op, с два реални параметъра и връщаща резултата от прилагането на операцията op към аргументите на функцията op. Така получаваме още едно обобщение на горните абстракции – функцията от по-висок ред accumulate (задача 82).

**Задача 82.** Да се напише програма, която реализира следната математическа абстракция:

$$f(a) \otimes f(next(a)) \otimes f(next(next(a)) \otimes \ldots \otimes f(b)$$

където с ⊗ е означена произволна бинарна целочислена операция, а f и next имат смисъла, определен в предходните две задачи.

Програма Zad82.cpp решава задачата.

```
// Program Zad82.cpp
#include <iostream.h>
#include <math.h>
double accumulate(double (*) (double, double),
                  double, double, double,
                  double (*)(double), double (*) (double));
double plus(double, double);
double mult(double, double);
double next1(double);
double next2(double);
int main()
{cout << "a, b= ";
 double a, b;
 cin >> a >> b;
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
 cout << accumulate(plus, 0, a, b, cos, next1) << '\n';
 cout << accumulate(mult, 1, a, b, sin, next2) << '\n';
 return 0;
}

double accumulate(double (*op)(double, double),
                  double null_val, double a, double b,
                  double (*f)(double), double (*next)(double))
{double s = null_val;
 for (double i = a; i <= b +1e-14; i = next(i))
    s = op(s, f(i));
 return s;
}
```

```
double next1(double x)
{return x + 1;
}
double next2(double x)
{return x + 2;
}
double plus(double x, double y)
{return x + y;
}
double mult(double x, double y)
{return x * y;
}
```

Функцията accumulate е функция от по-висок ред. Нейните първи, пети и шести формални параметри са указатели към функции, задаващи съответно операцията op, терма f и стъпката next.

В тази програма са направени две обръщения към функцията accumulate, които намират:

```
cos(a) + cos(a+1) + cos(a+2) + ... + cos(b)
```

и

```
sin(a) * sin(a+2) *  sin(a+4) * ... * sin(b)
```

съответно.

Чрез accumulate, функциите sum и prod могат да се дефинират по следния начин:

```
double sum(double a, double b, double (*f)(double),
           double (*next)(double))
{return accumulate(plus, 0, a, b, f, next);
}
double prod(double a, double b, double (*f)(double),
            double (*next)(double))
{return accumulate(mult, 1, a, b, f, next);
}
```

където plus и mult са дефинирани в програма Zad82.cpp.

Използването на променливи, които са указатели към функции, усложнява записа на дефиницията на функция. Добре би било да дадем имена на типовете указател към функция и вместо дефиницията да използваме името на типа. Задаването на имена на типове може да се

осъществи чрез оператора typedef. На Фиг. 9.2 са дадени синтаксисът и семантиката на този оператор.

**Оператор typedef**

*Синтаксис*

typedef <тип> <име>;

където

- <тип> е дефиниция на тип;
- <име> е идентификатор, определящ името на новия тип.

*Семантика*

Определя <име> за синоним на типа от <тип>.

Фиг. 9.2 Оператор typedef

*Примери:*

```
typedef unsigned char BYTE; // BYTE е синоним на unsigned char
typedef double REAL;  // REAL е синоним на double
```

Задаването на алтернативно име на тип указател към функция чрез typedef се осъществява по аналогичен начин на дефиниране на променлива от тип указател към функция като новото име на типа заема мястото на променливата.

*Примери:*

1. typedef double(*mytype)(double);

определя mytype като синоним на типа double (*)(double);

2. typedef double(*newtype)(double, double);

определя newtype като синоним на типа double (*)(double, double).

Като използваме оператора typedef и дефинираме:

```
typedef double (*type1) (double, double);
typedef double (*type2) (double);
```

декларацията на функцията accumulate от Zad82.cpp:

```
double accumulate(double (*) (double, double),
                  double, double, double,
                  double (*)(double), double (*) (double));
```

може да се запише по следния начин:

```
double accumulate(type1, double, double, double, type2, type2);
```

**Задача 83.** Като се направи подходяща модификация на accumulate, да се напише програма, която по дадени естествено число n и реално число x, намира сумата:

$$\sum_{i=0}^{n} \frac{x^i}{i!}$$

Програма Zad83.cpp решава задачата. В нея a и b са 0 и n съответно. Термът f е:

$$f: i \longrightarrow \frac{x^i}{i!},$$

а стъпката се задава от:

$$next: i \longrightarrow i + 1.$$

Направена е модификация на функцията accumulate. Последната се налага заради промяната на типовете на a, b, на f и next.

```
// Program Zad83.cpp
#include <iostream.h>
#include <math.h>
typedef double (*type1) (double, double);
typedef double (*type2)(int);
typedef int (*type3) (int);
double x;
double accumulate(type1, double, int, int, type2, type3);
double f(int);
int next(int);
double sum(double, double);
int main()
{cout << "n= ";
 int n;
 cin >> n;
 if (!cin || n < 0)
 {cout << "Error! \n";
  return 1;
 }
 cout << "x= ";
```

```
 cin >> x;
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
 cout << accumulate(sum, 0, 0, n, f, next) << '\n';
 return 0;
}
double f(int i)
{int p = 1;
 for (int j = 1; j <= i; j++) p = p*j;
  return pow(x, i)/p;
}
int next(int x)
{return x + 1;
}
double sum(double x, double y)
{return x + y;
}
double accumulate(type1 op, double null_val,
                  int a, int b, type2 f, type3 next)
{double s = null_val;
 for (int i = a; i <= b; i = next(i))
    s = op(s, f(i));
 return s;
}
```

## 9.3 Функциите като върнати оценки

Функция може да върне като резултат указател към друга функция. Например, декларацията

```
int (*fun(int, int))(int*, int);
```

определя функцията fun с два цели аргумента и връщаща указател към функция от тип

```
int (*)(int*, int).
```

Ако зададем име на този тип чрез typedef, т.е.

```
typedef int (*fun-point)(int*, int);
```

този запис може да се опрости до:

```
fun-point fun(int, int);
```

**Задача 84.** Да се напише програма, която по зададено реално число x и символ (a, b, c или d) избира за изпълнение функция, определена чрез зависимостта:

$$y = \begin{cases} \sin(x) & \longrightarrow a \\ \cos(x) & \longrightarrow b \\ \exp(x) & \longrightarrow c \\ \log(x) & \longrightarrow d. \end{cases}$$

Програма Zad84.cpp решава задачата.

```
// Program Zad84.cpp
#include <iostream.h>
#include <math.h>
typedef double (*f_type)(double);
f_type table(char ch)
{switch(ch)
 {case 'a': return sin; break;
  case 'b': return cos; break;
  case 'c': return exp; break;
  case 'd': return log; break;
  default: cout << "Error! \n"; return tan;
 }
}
int main()
{char ch;
 cout << "ch= ";
 cin >> ch;
 if (ch < 'a' || ch >'d') cout << "Incorrect input! \n";
 else
 {double x;
  cout << "x= ";
  cin >> x;
```

```
    cout << table(ch)(x) << '\n';
   }
   return 0;
  }
```

Илюстрираните в тази част възможности на езика C++ показват, че данните от тип функции съществено не се отличават от другите видове данни. Това показва високата степен на унифицираност в езика и води до увеличаване на изразителната му сила.

## Задачи

**Задача 1.** Като използвате функциите от по-висок ред sum и prod, намерете:

$$S = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{3!} + \ldots + (-1)^n \frac{x^{(2n+1)}}{(2n+1)!}$$

$$S = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$$

**Задача 2.** Като използвате функцията от по-висок ред prod, намерете:

а) $x^n$, където x е дадено реално, а n – дадено естествено число.

б) n!, където n е дадено естествено число.

в) броят на вариациите от n елемента от k-ти клас (n и k са дадени естествени числа, $0 \le k \le n$).

г) броят на комбинациите от n елемента от k-ти клас (n и k са дадени естествени числа, $0 \le k \le n$).

## Допълнителна литература


1. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.

2. Д. Луис, C/C++ бърз справочник, ИНФОДАР, София, 1998.

3. М. Тодорова, Езици за функционално и логическо програмиране. Функционално програмиране, СОФТЕХ, София, 1998.

4. B. Stroustrup, C++ Programming Language. Third Edition, Addison - Wesley, 1997.

# 10

# Рекурсия

## 10.1 Рекурсивни функции в математиката

Ако в дефиницията на някаква функция се използва самата функция, дефиницията на функцията се нарича **рекурсивна**.

*Примери:*

а) Ако n е произволно естествено число, следната дефиниция на функцията факториел

$$n! = \begin{cases} 1, & n = 0 \\ n.(n-1)!, & n > 0 \end{cases}$$

е рекурсивна. Условието при n = 0 не съдържа обръщение към функцията факториел и се нарича **гранично**.

б) Функцията за намиране на най-голям общ делител на две естествени числа a и b може да се дефинира по следния рекурсивен начин:

$$\gcd(a, b) = \begin{cases} a, & a = b \\ \gcd(a-b, b), & a > b \\ \gcd(a, b-a), & a < b. \end{cases}$$

Тук граничното условие е условието при a = b.

в) Ако x е реално, а n – цяло число, функцията за степенуване може да се дефинира рекурсивно по следния начин:

$$x^n = \begin{cases} x.x^{n-1}, & n>0 \\ 1, & n=0 \\ \dfrac{1}{x^{-n}}, & n<0. \end{cases}$$

В този случай граничното условие е условието при n = 0.

г) Редицата от числата на Фибоначи

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

се дефинира рекурсивно по следния начин:

$$fib(n) = \begin{cases} 0, & n=0 \\ 1, & n=1 \\ fib(n-1)+fib(n-2), & n>1. \end{cases}$$

В този случай имаме две гранични условия – при n = 0 и при n = 1.

Рекурсивната дефиниция на функция може да се използва за намиране стойността на функцията за даден допустим аргумент.

*Примери:*

а) Като се използва рекурсивната дефиниция на функцията факториел може да се намери факториелът на 4. Процесът за намирането му преминава през разширение, при което операцията умножение се отлага, до достигане на граничното условие 0! = 1. Следва свиване, при което се изпълняват отложените операции.

4! =

4.3! =

4.3.2! =

4.3.2.1! =

4.3.2.1.0! =

4.3.2.1.1 =

4.3.2.1 =

4.3.2 =

4.6 =

24

б) Като се използва рекурсивната дефиниция на функцията gcd, може да се намери gcd(35, 14).

gcd(35, 14) =

gcd(21, 14) =

gcd(7, 14) =

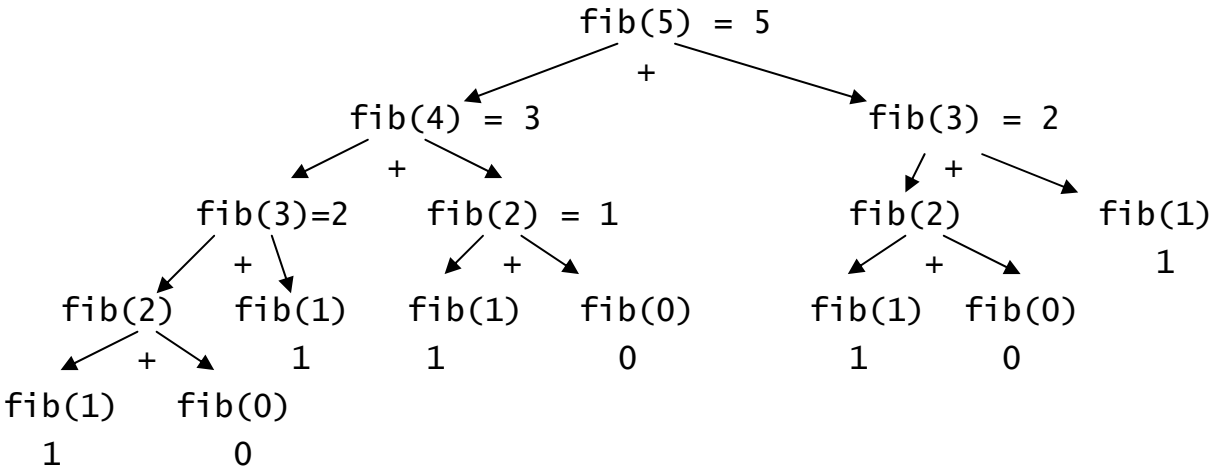
gcd(7, 7) = 7.

В този случай няма разширяване, нито пък свиване. Двата аргумента на gcd играят ролята на натрупващи параметри. Те се променят докато се достигне граничното условие.

в) Като се използва рекурсивната дефиниция на функцията за степенуване може да се намери стойността на $2^{-4}$.

$$2^{-4} =$$
$$\frac{1}{2^4} =$$
$$\frac{1}{2.2^3} =$$
$$\frac{1}{2.2.2^2} =$$
$$\frac{1}{2.2.2.2.2^0} =$$
$$\frac{1}{2.2.2.2.1} =$$
$$\frac{1}{2.2.2.2}$$
$$\frac{1}{2.2.4} =$$
$$\frac{1}{2.8} = \frac{1}{16}.$$

г) Чрез рекурсивната дефиниция на функцията, генерираща редицата на Фибоначи, може да се намери fib(5).



В този случай намирането на fib(5) генерира дървовиден процес. Операцията + се отлага. Забелязваме много повтарящи се изчисления, например fib(0) се пресмята 3 пъти, fib(1) – 5 пъти, fib(2) - 3 пъти, fib(3) – 2 пъти, което илюстрира неефективността на този начин за пресмятане.

## 10.2 Рекурсивни функции в C++

Известно е, че в тялото на всяка функция може да бъде извикана друга функция, която е дефинирана или е декларирана до момента на извикването й. Освен това, в C++ е вграден т. нар. механизъм на рекурсия – разрешено е функция да вика в тялото си самата себе си.

Функция, която се обръща пряко или косвено към себе си, се нарича рекурсивна. Програма, съдържаща рекурсивна функция е **рекурсивна**.

Чрез примери ще илюстрираме описанието, обръщението и изпълнението на рекурсивна функция.

**Задача 85.** Да се напише рекурсивна програма за намиране на m! (m е дадено естествено число).

Програма Zad85.cpp решава задачата.

```
// Program Zad85.cpp
#include <iostream.h>
int fact(int);
int main()
{cout << "m= ";
 int m;
 cin >> m;
 if (!cin || m < 0)
 {cout << “Error! \n”;
  return 1;
 }
 cout << m << "!= " << fact(m) << '\n';
 return 0;
}
int fact(int n)
```

```
{if (n == 0) return 1;
 else return n * fact(n-1);
}
```

В тази програма е описана рекурсивната функция fact, която приложена към естествено число връща факториела на това число. Стойността на функцията се определя посредством обръщение към самата функция в оператора return n * fact(n-1);. Запазената дума else в оператора

```
if (n == 0) return 1;
else return n * fact(n-1);
```

е излишна заради оператора return 1; пред нея. Използвана е с цел увеличаване на читаемостта на функцията.

*Изпълнение на програмата*

Изпълнението започва с изпълнение на главната функция. Фрагментът

```
cout << "m= ";
int m;
cin >> m;
```

въвежда стойност на променливата m. Нека за стойност на m е въведено 4. В резултат в стековата рамка на main, отделените 4В за променливата m се инициализират с 4. След това се изпълнява операторът

```
cout << m << "!= " << fact(m) << '\n';
```

За целта трябва да се пресметне стойността на функцията fact(m) за m равно на 4, след което получената стойност да се изведе. Обръщението към функцията fact е илюстрирано по-долу:

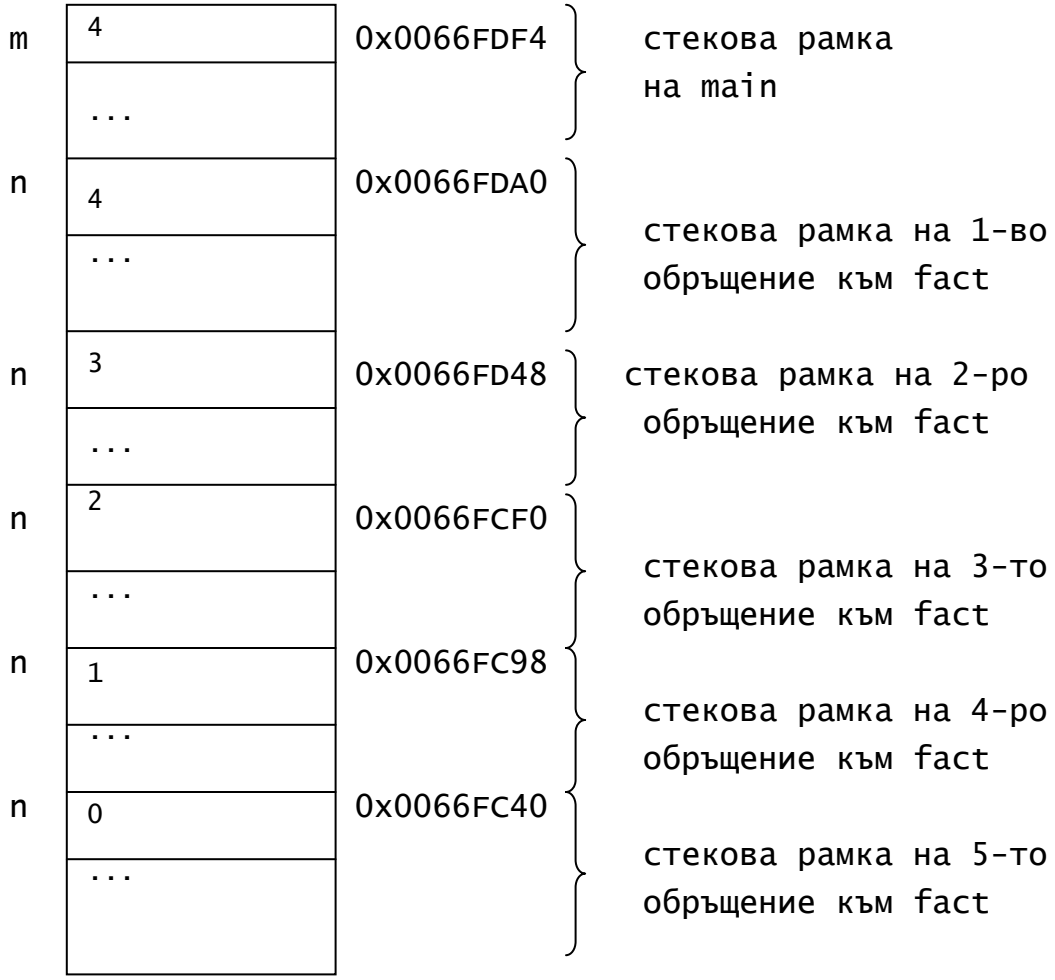
Генерира се стекова рамка за това обръщение към функцията fact. В нея се отделят 4В ОП за фактическия параметър n, в която памет се откопирва стойността на фактическия параметър m и започва изпълнението на тялото на функцията. Тъй като n е различно от 0, изпълнява се операторът

```
return n * fact(n-1);
```

при което трябва да се намери fact(n-1), т.е. fact(3). По такъв начин преди завършването на първото обръщение към fact се прави второ обръщение към тази функция. За целта се генерира нова стекова рамка на функцията fact, в която за формалния параметър n се откопирва стойност 3. Тялото на функцията fact започва да се изпълнява за втори

път (Временно спира изпълнението на тялото на функцията, предизвикано от първото обръщение към нея).

По аналогичен начин възникват още обръщения към функцията fact. При последното от тях, стойността на формалния параметър n е равна на 0. Получава се:




При петото обръщение към fact стойността на n е равна на 0. В резултат, изпълнението на това обръщение завършва и за стойност на fact се получава 1. След това последователно завършват изпълненията на останалите обръщения към тялото на функцията. При всяко изпълнение на тялото на функцията се определя съответната стойност на функцията fact. След завършването на всяко изпълнение на функцията fact, отделената за fact стекова рамка се освобождава. В крайна сметка в главната програма се връща 24 - стойността на 4!, която се извежда върху екрана.

В този случай, рекурсивното дефиниране на функцията факториел не е подходящо, тъй като съществува лесно итеративно решение.

**Препоръка:** Ако за решаването на някаква задача може да се използва итеративен алгоритъм, реализирайте го. Не се препоръчва винаги използването на рекурсия, тъй като това води до загуба на памет и време.

В тази глава няма да спазим препоръката, тъй като целта ни е придобиване на умения за рекурсивно дефиниране на функции.

**Задача 86.** Да се напише програма, която въвежда от клавиатурата записана без грешка формула от вида

```
<формула> ::=  <цифра> |
               (<формула><знак><формула>)
<знак> ::= + | - | *
<цифра> ::= 0 | 1 | 2 | ...| 9.
```

Програмата да намира и извежда стойността на въведената формула (Например 8 --> 8; ((2-6)*4) --> -16).

Програма Zad86.cpp решава задачата. В нея е дефинирана рекурсивната функция formula, реализираща рекурсивната дефиниция на <формула>. Ще отбележим, че случаите на оператора switch не завършват с оператора break;. Това е така заради използването на оператора return в края на всеки случай.

```
// Program Zad86.cpp
#include <iostream.h>
int formula();
int main()
{cout << formula() << "\n";
 return 0;
}
int formula()
{char c, op;
 int x, y;
 cin >> c;                  // c е ‘(‘ или цифра
 // <формула> ::= <цифра>
```

```
 if (c >= '0' && c <= '9') return (int)c – (int)'0';
 else
 // <формула> ::= (<формула><знак><формула>)
 {x = formula();
  cin >> op;
  y = formula();
  cin >> c;                // прескачане на ‘)’
  switch (op)
  {case '+': return x + y;
   case '-': return x - y;
   case '*': return x * y;
   default: cout << "Error! \n"; return 111;
  }
 }
}
```

Забелязваме простотата и компактността на записа на рекурсивните функции. Това проличава особено при работа с динамичните структури: свързан списък, стек, опашка, дърво и граф.

Основен недостатък е намаляването на бързодействието поради загуба на време за копиране на параметрите им в стека. Освен това се изразходва повече памет, особено при дълбока степен на вложеност на рекурсията.

## Задачи върху рекурсия

**Задача 87.** Да се напише рекурсивна програма, която намира най-големия общ делител на две естествени числа.

Програма Zad87.cpp решава задачата.

```
// Program Zad87.cpp
#include <iostream.h>
int gcd(int, int);
int main()
{cout << "a, b= ";
 int a, b;
 cin >> a >> b;
```

```
 if (!cin || a < 1 || b < 1)
 {cout << "Error! \n";
  return 1;
 }
 cout << "gcd{" << a << ", " << b << "}= " <<  gcd(a, b) << "\n";
 return 0;
}
int gcd(int a, int b)
{if (a == b) return a;
 if (a > b) return gcd(a-b, b);
 return gcd(a, b-a);
}
```

**Задача 88.** Като се използва рекурсивната дефиниция на функцията за степенуване да се напише програма, която по дадени a реално и k – цяло число, намира стойността на $a^k$.

Програма Zad88.cpp решава задачата.

```
// Program Zad88.cpp
#include <iostream.h>
double pow(double, int);
int main()
{cout << "a= ";
 double a;
 cin >> a;
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
 cout << "k= ";
 int k;
 cin >> k;
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
```

```
 cout << "pow{" << a << ", " << k << "}= " << pow(a, k) << "\n";
 return 0;
}
double pow(double x, int n)
{if (n == 0) return 1;
 if (n > 0) return x * pow(x, n-1);
 return 1.0/pow(x, -n);
}
```

**Задача 89.** Квадратна мрежа с k реда и k стълба (1 ≤ k ≤ 20) има два вида квадратчета - бели и черни. В черно квадратче може да се влезе, но не може да се излезе. От бяло квадратче може да се премине във всяко от осемте му съседни, като се прекоси общата им страна или връх. Да се напише програма, която ако са дадени произволна мрежа с бели и черни квадратчета и две произволни квадратчета - начално и крайно, определя дали от началното квадратче може да се премине в крайното.

*Анализ на задачата:*

а) Ако началното квадратче не е в мрежата, приемаме, че не може да се премине от началното до крайното квадратче.

б) Ако началното квадратче съвпада с крайното, приемаме, че може да се премине от началното до крайното квадратче.

в) Ако началното и крайното квадратчета са различни и началното квадратче е черно, не може да се премине от него до крайното квадратче.

г) Във всички останали случаи, от началното квадратче може да се премине до крайното тогава и само тогава, когато от някое от съседните му квадратчета (в хоризонтално, във вертикално или диагонално направление), може да се премине до крайното квадратче.

Програма Zad89.cpp решава задачата.

*Представяне на данните:*

Мрежата ще представим чрез квадратна матрица от цели числа mr[k x k] (1 ≤ k ≤ 20) и ще реализираме чрез съответен двумерен масив. При това mr[i][j] е равно на 0, ако квадратче (i, j) е бяло и е равно на

1, ако е квадратче (i, j) е черно (0 ≤ i ≤ k-1, 0 ≤ j ≤ k-1). Нека началното квадратче е от ред x и стълб y, а крайното квадратче е от ред m и стълб n.

В програмата Zad89.cpp е дефинирана рекурсивната функция way. Тя връща стойност true, ако от квадратче (x, y) може да се премине до квадратче (m, n) и false - в противен случай. За да се избегне зацикляне (връщане в началното квадратче от всяко съседно), се налага преди рекурсивните обръщения към way, да се промени стойността на mr[x][y] като квадратчето (x, y) се направи черно.

```
// Program Zad89.cpp
#include <iostream.h>
int mr[20][20];
int k, m, n;
bool way(int, int);
void writemr(int, int [][20]);
int main()
{cout << "k from [1, 20] = ";
 do
 {cin >> k;
 }while (k < 1 || k > 20);
 int x, y;
 do
 {cout << "x, y = ";
  cin >> x >> y;
 } while (x < 0 || x > k-1 || y < 0 || y > k-1);
 do
 {cout << "m, n = ";
  cin >> m >> n;
 } while (m < 0 || m > k-1 || n < 0 || n > k-1);
 for (int i = 0; i <= k-1; i++)
 for (int j = 0; j <= k-1; j++)
   cin >> mr[i][j];
 cout << "\n";
 writemr(k, mr);
 if (way(x, y)) cout << "yes \n";
 else cout << "no \n";
```

```
 return 0;
}
bool way(int x, int y)
{if (x < 0 || x > k-1 || y < 0 || y > k-1) return false;
 if (x == m && y == n) return true;
 if (mr[x][y] == 1) return false;
 mr[x][y] = 1;
 bool b = way(x+1,  y) || way(x-1, y)  ||
          way(x,  y+1) || way(x, y-1)  ||
          way(x-1,y-1) || way(x-1, y+1)||
          way(x+1,y-1) || way(x+1, y+1);
 mr[x][y] = 0;
 return b;
}
void writemr(int k, int mr[][20])
{for (int i = 0; i <= k-1; i++)
  {for (int j = 0; j <= k-1; j++)
     cout << mr[i][j];
   cout << '\n';
  }
}
```

**Задача 90.** Да се напише рекурсивен вариант на функцията от по-висок ред accumulate.

Функцията accumulate, дефинирана по-долу, решава задачата. Тя използва дефинициите на типовете:

```
typedef double (*type1) (double, double);
typedef double (*type2) (double);

double accumulate(type1 op, double null_val, double a, double b,
                  type2 f, type2 next)
{if (a > b + 1e-14) return null_val;
 return op(f(a),
                accumulate(op, null_val, next(a), b, f, next));
}
```

**Задача 91.** Да се напише рекурсивна функция double min(int n, double* x), която намира минималния елемент на редицата $x_0$, $x_1$, ..., $x_{n-1}$.

*първо решение:*

```
double min(int n, double* x)
{double b;
 if (n == 1) return x[0];
 b =  min(n-1, x);
 if (b < x[n-1]) return b;
 return x[n-1];
 }
```

*второ решение:*

```
double min(int n, double* x)
{double b;
 if (n == 1) return x[0];
 b =  min(n-1, x+1);
 if (b < x[0]) return b;
 return x[0];
 }
```

**Задача 92.** Да се напише рекурсивна функция, която проверява дали елементът x принадлежи на редицата $a_0$, $a_1$, ..., $a_{n-1}$.

*първо решение:*

```
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
```

*второ решение:*

```
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[n-1] || member(x, n-1, a);
}
```

**Задача 93.** Да се напише рекурсивна функция, която проверява дали редицата $x_0$, $x_1$, ..., $x_{n-1}$ е монотонно растяща.

*първо решение:*

```
bool monincr(int n, double* x)
{if (n == 1) return true;
 return x[n-2] <= x[n-1] && monincr(n-1, x);
}
```

*второ решение:*

```
bool monincr(int n, double* x)
{if (n == 1) return true;
 return x[0] <= x[1] && monincr(n-1, x+1);
}
```

**Задача 94.** Да се напише рекурсивна функция, която проверява дали редицата $a_0$, $a_1$, ..., $a_{n-1}$ се състои от различни елементи.

*първо решение:*

```
bool differ(int n, int* a)
{if (n == 1) return true;
 return !member(a[0], n-1, a+1) && differ(n-1, a+1);
}
```

*второ решение:*

```
bool differ(int n, int* a)
{if (n == 1) return true;
 return !member(a[n-1], n-1, a) && differ(n-1, a);
}
```

**Задача 95.** Дадена е квадратна мрежа от клетки, всяка от които е празна или запълнена. Запълнените клетки, които са свързани, т.е. имат съседни в хоризонтално, вертикално или диагонално направление, образуват област. Да се напише програма, която намира броя на областите и размера (в брой клетки) на всяка област.

Програма Zad95.cpp решава задачата.

*Представяне на данните:*

Мрежата ще представим чрез квадратна матрица от цели числа mr[n x n] (1 ≤ n ≤ 20) и ще реализираме чрез съответен двумерен масив. При това mr[i][j] е 1 ако квадратче (i, j) е запълнено и 0 – в противен случай (0 ≤ i ≤ n-1, 0 ≤ j ≤ n-1).

*Анализ на задачата:*

Ще дефинираме функция broy, която преброява клетките в областта, съдържаща дадена клетка (x, y). Функцията има два параметъра x и y - координатите на точката и реализира следния алгоритъм:

а) Ако клетката с координати (x, y) е извън мрежата, приемаме, че броят на клетките в областта е равен на 0.

б) В противен случай, ако клетката с координати (x, y) е празна, приемаме, че броят е равен на 0.

в) В останалите случаи, броят на клетките в областта е равен на сумата от 1 и броя на клетките на всяка област, на която принадлежат осемте съседни клетки на клетката (x, y).

От подточка в) следва, че функцията broy е рекурсивна. За да избегнем зацикляне и многократно преброяване, трябва преди рекурсивното обръщение на направим клетката (x, y) празна.

```
// Program Zad95.cpp;
#include <iostream.h>
int mr[20][20];
int n;
int broy(int x, int y)
{if (x < 0 || x > n-1 || y < 0 || y > n-1) return 0;
 if (mr[x][y] == 0) return 0;
 mr[x][y] = 0;
 return 1 + broy(x-1, y+1) + broy(x, y+1)
          + broy(x+1, y+1) + broy(x+1,y)
          + broy(x+1, y-1) + broy(x, y-1)
          + broy(x-1, y-1) + broy(x-1, y);
}
int main()
{cout << "mreja: \n”;
 do
 {cout << "n= ";
  cin >> n;
```

```
 } while (n < 1 && n > 20);
 int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
    cin >> mr[i][j];
int br = 0;
for (i = 0; i <= n-1; i++)
 for (j = 0; j <= n-1; j++)
  if (mr[i][j] == 1)
  {br++;
   cout << "size of the " <<  br << " th location is equal to "
        << broy(i, j) << endl;
  }
return 0;
}
```

**Задача 96.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която установява дали съществува път между два произволно зададени града (Приемаме, че ако от град i до град j има път, то има път и от град j до град i).

*Анализ на задачата:*

Ще дефинираме рекурсивната булева функция way, която зависи от два параметъра i и j, показващи номерата на градовете, между които се проверява дали съществува път. Функцията реализира следния алгоритъм:

а) Ако i = j, съществува път от град i до град j.

б) Ако $i \ne j$ и има пряк път от град i до град j, има път между двата града.

в) В останалите случаи има път от град i до град j, тогава и само тогава, когато съществува град k, с който град i е свързан с пряк път и от който до град j има път.

Програма Zad96.cpp решава задачата.

```
// Program Zad96.cpp;
#include <iostream.h>
```

```
int arr[10][10];
int n;
bool way(int i, int j)
{if (i == j) return true;
 if (arr[i][j] == 1) return true;
 bool b = false;
 int k = -1;
 do
 {k++;
  if (arr[i][k] == 1)
  {arr[i][k] = 0; arr[k][i] = 0;
   b = way(k, j);
   arr[i][k] = 1; arr[k][i] = 1;
   }
 } while (!b && k <= n-2);
 return b;
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i
         << " and " << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
```

```
 } while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (way(i, j)) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

**Задача 97.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която намира пътя между два произволно зададени града в случай, че път между тях съществува.

*Анализ на задачата:*

Процедурата foundway намира един път от град i до град j в случай, че път съществува, т.е. ако way(i, j) има стойност true. Пътят се записва в едномерния масив int x[100], а дължината му - в променливата s.

Програма Zad97.cpp решава задачата. В нея е пропусната дефиницията на функцията way. При обръщение към функцията foundway параметърът-псевдоним s трябва да се свърже с параметър, инициализиран с –1.

```
// Program Zad97.cpp
#include <iostream.h>
int arr[10][10];
int n;
bool way(int i, int j)
...
void foundway(int i, int j, int& s, int x[])
{s++;
 x[s] = i;
 if (i != j)
  if (arr[i][j] == 1)
  {s++;
   x[s] = j;
  }
  else
```

```
  {bool b = false;
   int k = -1;
   do
   {k++;
    if (arr[i][k] == 1) b = way(k, j);
   } while (!b);
   arr[i][k] = 0; arr[k][i] = 0;
   foundway(k, j, s, x);
   arr[i][k] = 1; arr[k][i] = 1;
  }
}
int main()
{int p = -1;
 int a[100];
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
   cin >> arr[i][j];
   arr[j][i] = arr[i][j];
   }
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
 } while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (way(i, j))
 {foundway(i, j, p, a);
  p++;
```

```
    for (int l = 0; l <= p-1; l++) cout << a[l] << "  " ;
    cout << endl;
   }
   else cout << "no \n";
   return 0;
  }
```

**Задача 98.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише булева функция fixway(int i, int j, int p), която установява дали  съществува път от град i до град j с дължина p ($p \ge 1$).

*Анализ на задачата:*

Булевата функция fixway реализира следния алгоритъм:

а) Ако p = 1, има път от град i до град j с дължина p ако има пряк път между двата града.

б) Ако p > 1, има път от град i до град j с дължина p тогава и само тогава, когато съществува град k, с който град i е свързан с пряк път и от който до град j има път с дължина p-1.

Следващият програмен фрагмент дефинира само функцията fixway.

```
bool fixway(int i, int j, int p)
{if (p == 1) return arr[i][j] == 1; else
 {bool b = false;
  int k = -1;
  do
  {k++;
   if (arr[i][k] == 1) b = fixway(k, j, p-1);
  } while (!b && k <= n-2);
  return b;
 }
}
```

Тази функция извършва проверка за съществуване на цикличен път от един до друг връх с указана дължина.

*Пример:* Ако имаме само два града, означени с 0 и 1 и те са свързни с пряк път, fixway(0, 1, 3) ще отговори с true.

Ако търсим съществуването само на ациклични пътища, ще използваме модификацията на горната функция, дадена по-долу.

```
bool fixway(int i, int j, int p)
{bool b;
 int k;
 if (p == 1) return arr[i][j] == 1;
 b = false;
 k = -1;
 do
 {k++;
  if (arr[i][k] == 1)
  {arr[i][k] = 0; arr[k][i] = 0;
   b = fixway(k, j, p-1);
   arr[i][k] = 1; arr[k][i] = 1;
  }
 } while (!b && k <= n-2);
 return b;
}
```

**Задача 99.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише процедура foundfixway(int i, int j, int p, int s&, int* x), която намира пътя от град i до град j с дължина p в случай, че такъв съществува.

Програма Zad99.cpp решава задачата. За краткост, дефиницията на функцията fixway е пропусната. Пътят е запомнен в масива x, а параметърът s съдържа текущата му дължина.

```
// Program Zad99.cpp
#include <iostream.h>
int arr[10][10];
int n;
bool fixway(int i, int j, int p)
...
void foundfixway(int i, int j, int p, int& s, int* x)
```

```
{s++;
 x[s] = i;
 if (p == 1)
 {s++;
  x[s] = j;
 }
 else
 {bool b = false;
  int k = -1;
  do
  {k++;
   if (arr[i][k] == 1) b = fixway(k, j, p-1);
  } while (!b);
  foundfixway(k, j, p-1, s, x);
 }
}
int main()
{int p = -1;
 int a[100];
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
 int l;
 do
```

```
 {cout << "start and final towns, and len between them: ";
  cin >> i >> j >> l;
 } while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (fixway (i, j, l))
 {foundfixway(i, j, l, p, a);
  for (int m = 0; m <= l; m++) cout << a[m] << "  " ;
  cout << endl;
 }
 else cout << "no \n";
 return 0;
}
```

## Задачи

**Задача 1.** Да се напише рекурсивна функция, която намира стойността на функцията на Акерман Ack(m, n), дефинирана за m ≥ 0 и n ≥ 0 по следния начин:

Ack(0, n) = n+1

Ack(m, 0) = Ack(m-1, 1), m > 0

Ack(m, n) = Ack(m-1, Ack(m, n-1)), m > 0, n > 0.

**Задача 2.** Да се напише рекурсивна функция, която установява, дали в записа на естественото числа n се съдържа цифрата k.

**Задача 3.** Да се напише рекурсивна програма, която намира стойността на полинома

$$P_n(x) = a_0x^{n-1} + a_1x^{n-2} + ... + a_{n-2}x + a_{n-1}$$

където x и $a_i$ (0 ≤ i ≤ n) са дадени реални числа.

**Задача 4.** Да се напише рекурсивна функция, която пресмята корен квадратен от x, x ≥ 0, по метода на Нютон.

**Задача 5.** Да се напише рекурсивна функция, която добавя елемент в сортиран масив, като запазва наредбата на елементите.

**Задача 6.** Да се напише рекурсивна функция, която изключва елемент от сортиран масив, като запазва наредбата на елементите.

**Задача 7.** Дадена е мрежа от m x n квадратчета, като за всяко квадратче е определен цвят - бял или черен. Път ще наричаме редица от

съседни във вертикално или хоризонтално направление квадратчета с един и същ цвят. Област ще наричаме множество от квадратчета с един и същ цвят между всеки две, от които има път. Дадено е квадратче. Да се определи:

а) броят на квадратчетата от областта, в която се съдържа даденото квадратче.

б) броят на областите с цвят, съвпадащ с цвета на даденото квадратче.

в) броят на областите с цвят, различен от цвета на даденото квадратче.

г) броят на квадратчетата с цвят, съвпадащ с цвета на даденото квадратче, които не са в една област с него.

**Задача 8.** Дадено е множество от n града и за всеки два от тях е определено дали са свързани с път или не. Едно множество от градове ще наричаме пълно, ако всеки два различни града, принадлежащи на множеството, са свързани с път. Да се напише програма, която по дадено k, k < n, извежда всички пълни множества, състоящи се от k на брой града.

**Задача 9.** Дадено е множество от n града и за всеки два от тях е определено дали са свързани с път или не. Едно множество от градове ще наричаме независимо, ако всеки два различни града, принадлежащи на множеството, не са свързани с път. Да се напише програма, която по дадено k, k < n, извежда всички независими множества, състоящи се от k на брой града.

**Задача 10.** Да се напише програма, която намира стойността на произволен израз от вида:

<израз> ::= <цяло_число>|
(<израз> * <израз>).

**Задача 11.** Да се напише програма, която намира стойността на произволен израз от вида:

<израз> ::= <цяло_число>|
(<израз> ^ <израз>),

където с ^ е означена операцията степенуване.

**Задача 12.** Да се напише програма, която въвежда от клавиатурата без грешка булев израз от вида

<булев_израз> ::= t|f|

```
                    <операция>(<операнди>)
<операция> ::= n | a | o
<операнди> ::= <операнд>|
               <операнд>, <операнди>
<операнд> ::= <булев_израз>,
```

където t и f означават истина и лъжа съответно, **n** има само един операнд, а **a** и **o** могат да имат произволен брой операнди и означават съответно логическо отрицание, конюнкция и дизюнкция. Програмата намира и извежда стойността на булевия израз.

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
3. М. Тодорова, Програмиране на Паскал, Полипринт Враца, София, 1993.
4. М. Тодорова, Езици за функционално и логическо програмиране – функционално програмиране, СОФТЕХ, София, 1998.

# 11

# Структури

## 11.1 Структура от данни запис

*Логическо описание*

Записът е съставна статична структура от данни, която се определя като крайна редица от фиксиран брой елементи, които могат да са от различни типове. Достъпът до всеки елемент от редицата е пряк и се осъществява чрез име, наречено **поле на записа**.

*Физическо представяне*

Полетата на записа се представят последователно в паметта.

*Примери:*

1. Данните за студент от една група (име, адрес, факултетен номер, оценки по изучаваните предмети) могат да се зададат чрез запис с четири полета.

2. Данните за книга от библиотека (заглавие, автор, година на издаване, издателство, цена) могат да се зададат чрез запис с пет полета.

3. Комплексно число може да се зададе чрез запис с две реални полета.

В езика C++ записите се реализират чрез структури. Ще разгледаме последните в развитие. Отначало ще опишем възможностите им на ниво – език C.

## 11.2 Дефиниране и използване на структури

Една структура се определя чрез имената и типовете на съставящите я полета. Фиг. 11.1 дава непълна дефиниция на структура.

```
Дефиниция на структура
<дефиниция_на_структура> ::= struct <име_на_структура>
            {<дефиниция_на_полета>;
             {<дефиниция_на_полета>;}опц
            };
<име_на_структура> ::= <идентификатор>
<дефиниция_на_полета> ::= <тип> <име_на_поле>{, <име_на_поле>}опц
<тип> ::= <име_на_тип>|<дефиниция_на_тип>
<име_на_поле> ::= <идентификатор>
```

Фиг. 11.1 Дефиниция на структура

Структурите, дефинирани по този начин, могат да се използват като типове данни. Имената на полетата в рамките на една дефиниция на структура трябва да са *различни* идентификатори.

*Примери:*

1. struct complex
   {double re, im;};

задава структура с име complex с две полета с имена re и im от тип double. Чрез нея се задават комплексните числа.

2. struct book
   {char name[41], author[31];
    int year;
    double price;
   };

задава структура с име book с четири полета: *първо поле* с име name от тип символен низ с максимална дължина 40 и определящо името на книгата; *второ поле* с име author от тип символен низ с максимална дължина 30, определящо името на автора на книгата; *трето поле* с име year от тип int, определящо годината на издаване и *четвърто поле* с

име price от тип double и определящо цената на книгата. Чрез тази структура се задава информация за книга.

3. struct student
   {int facnum;
    char name[36];
    double marks[30];
   };

задава структура с име student и с три полета: *първо поле* с име facnum от тип int, означаващо факултетния номер на студента; *второ поле* с име name от тип символен низ с максимална дължина 35, определящо името на студента и *трето поле* с име marks от тип реален масив с 30 компоненти и означаващо оценките от положените изпити.

Възможно е за име на структура, на нейно поле и на произволна променлива на програмата да се използва един и същ идентификатор. Но тъй като това намалява читаемостта на програмата, засега не препоръчваме използването му.

Допуска се влагане на структури, т.е. поле на структура може да е от тип структура.

*Пример:* Допустими са дефинициите:

```
struct xx
{int a, b, c;
};
struct pom
{int a;
 double b;
 char c;
 xx d;
};
```

Не е възможно обаче поле на структура да е от тип, съвпадащ с името на структурата.

*Пример:* Не е допустима дефиниция от вида

```
struct xxx{
   xxx member;    // опит за рекурсивна дефиниция
};
```

тъй като компилаторът не може да определи размера на xxx. Допустима е обаче дефиницията:

```
struct xxx{
   xxx* member;
};
```

**Допълнение:** За да е възможно две дефиниции на структури да се обръщат една към друга, е необходимо пред дефинициите им да се постави декларацията на втората структура. Например, ако искаме дефиницията на структурата list да използва дефиницията на структурата link и обратно, ще трябва да ги подредим по следния начин:

```
struct list;      // декларация на втората структура
struct link{
   link* pred;
   link* succ;
   list* member;
};
struct list{
   link* head;
};
```

Дефиницията на структура не предизвиква отделянето на памет за съхраняване на компонентите й. Може да се постави извън функция, в началото на функция или в началото на блок. Местоположението на дефиницията определя областта на името на структурата – съответно за всички функции след дефиницията, в рамките на функцията и в рамките на блока. Най-често дефиницията се задава пред първата функция на програмата и така става достъпна за всички функции.

Тъй като дефинирането на структура чрез задаване на името на структурата определя нов тип данни, ще определим множеството от стойности и операциите и вградените функции, свързани с него.

**Множество от стойности**

Множеството от стойностите на една структура се състои от всички крайни редици от по толкова елемента, колкото са полетата й, като всеки елемент е от тип, съвместим с типа на съответното поле.

*Примери:*

1. Множеството от стойности на структурата complex се състои от всички двойки от реални числа.

2. Множеството от стойности на структурата book се състои от всички четворки от вида:

{char[41], char[31], int, double}.

3. Множеството от стойности на структурата student се състои от всички тройки от вида:

{int, char[36], double[30]}.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на дадена структура, е променлива от дадения тип структура. Променлива от тип структура се дефинира в областта на структурата по следния начин (Фиг. 11.2):

**Дефиниция на променлива от тип структура**

<дефиниция_на_променлива_от_тип_структура> ::=
[struct]$_{опц}$ <име_на_структура>
<променлива> [={<редица_от_изрази>}]$_{опц}$
{,<променлива> [={<редица_от_изрази>}]$_{опц}$}$_{опц}$ ;|
struct {<дефиниция_на_полета>;
{<дефиниция_на_полета>;}$_{опц}$
}<променлива> [={<редица_от_изрази>}]$_{опц}$
{,<променлива> [={<редица_от_изрази>}]$_{опц}$}$_{цоп}$ ;
<променлива> ::= <идентификатор>
<редеица_от_изрази> ::= <израз>|
<израз>,<редица_от_изрази>

Фиг. 11.2 Дефиниция на променлива от тип структура

В C++ използването на запазената дума struct не е задължително. Някои програмисти го използват заради яснотата на кода. Конструкцията {<редица_от_изрази>} предизвиква инициализация на дефинирана променлива. Изразите, изредени във фигурните скоби, се разделят със запетая. Всеки израз инициализира поле на структурата и трябва да е от тип, съвместим с типа на съответното поле. Ако са записани по-

малко изрази от броя на полетата, компилаторът допълва останалите с нулевите стойности за съответния тип на поле.

*Примери:*

```
complex z1, z2 = {5.6, -8.3}, z3;
book b1, b2, b3;
struct student s1 = {44505, “Ivan Ivanov”, {5.5, 6, 5, 6}};
struct
{int x;
 double y;
} p, q = {-2, -1.6};
pom y;
```

Последната дефиниция от примера по-горе дефинира променливите p и q като променливи от тип структурата

```
struct
{int x;
 double y;
}
```

която участва с дефиницията, а не с името си.

Достъпът до полетата на структура е пряк. Един начин за неговото осъществяване е чрез променлива от тип структурата, като променливата и името на полето на структурата се разделят с оператора точка (Фиг. 11.3). Получените конструкции са *променливи* от типа на полето и се наричат **полета на променливата** от тип структура или **член-данни на структурата**, свързани с променливата.

Операторът . е лявоасоциативен и има приоритет равен на този на () и [].

**Поле на структура**

```
 <поле_на_променлива_структура> ::=
      <променлива_структура>.<име_на_поле>
```

Фиг. 11.3 Поле на структура

*Примери:*

С променливите z1, z2, z3 се свързват променливите от тип double: z1.re, z1.im, z2.re, z2.im, z3.re и z3.im

а с b1, s1 и  y –

b1.name - от тип char [41], b1.author – от тип char [31],
b1.year - от тип int, b1.price - от тип double,
s1.facnum – от тип int, s1.name - от тип char [36],
s1.marks - от тип double [30], y.a - от тип int,
y.b - от тип double, y.c - от тип char,
y.d - от тип xx и
с полета y.d.a, y.d.b и y.d.c от тип int.

Дефинирането на променлива от тип структура свързва променливата с множеството от стойности на съответната структура. След дефинициите от примера по-горе, променливите z1, z2 и z3 се свързват с множеството от стойностите на типа complex. При това свързване z2 е свързано с комплексното число 5.6–i8.3 чрез инициализация. Свързването на z1 и z3 със съответни комплексни числа може да стане чрез инициализация, подобно на z2, чрез присвояване, например z1 = z2;, или чрез задаване на стойности на полетата на променливата, например

```
z3.re = 3.4;
z3.im = -30.5;
```

свързва z3 с комплексното число 3.4 –i30.5.

Освен това, дефинирането на променлива от тип структура предизвиква отделяне на определено количество памет за всяко поле на променливата. Последното се определя от типа на полето. Полетата се разполагат последователно в паметта. Обикновено всяко поле се разполага от началото на машинна дума. Полетата, които не изискват цяло число на брой машинни думи, не използват напълно отредената им памет. Този начин за подреждане на полетата на структура се нарича **изравняване до границата на машинна дума**. За реализацията Visual C++ 6.0 размерът на 1 машинна дума е 8в.

*Пример:* За променливите p и q, дефинирани по-горе, ще се отделят по 16, а не по 12 байта

ОП

| p.x | p.y | q.x | q.y |
|---|---|---|---|
| - | - | -2 | -1.6 |
| 8в | 8в | 8в | 8в |

**Допълнение:** Две структури, дефинирани по един и същ начин, са различни. Например, дефинициите

```
struct str1{
   int a;
   int b;
};
```

и

```
struct str2{
   int a;
   int b;
};
```

определят два различни типа. Дефинициите

```
str1 x;
str2 y = x;
```

предизвикват грешка заради смесване на типовете (x и y са от различни типове).

**Операции и вградени функции**

Операциите над структури зависят от реализацията на езика. По стандарт за всяка реализация са определени следните операции и вградени функции:

*а) над полетата на променливи от тип структура*

Всяко поле на променлива от тип структура е от някакъв тип. Всички операции и вградени функции, допустими за данните от този тип, са допустими и за съответното поле.

*б) над променливи от тип структура*

- Възможно е на променлива от тип структура да се присвои стойността на вече инициализирана променлива от същия тип структура или стойността на израз от същия тип.

*Пример:* Допустими са присвояванията

```
z3 = z2;
p = q;
```

- Възможно е формален параметър на функция, а също резултатът от изпълнението й, да са структури. Структури с големи размери обикновено се предават чрез указатели или псевдоними на структури. Така се спестяват ресурси. Освен това, тези начини за предаване са по-сигурни.

Ще илюстрираме използването на структурите чрез следния пример.

**Задача 100.** Да се напише програма, която:

а) въвежда факултетните номера, имената и оценките по 5 предмета на студентите от една група;

б) извежда в табличен вид въведените данни;

в) сортира в низходящ ред по среден успех данните;

г) извежда сортираните данни, като за всеки студент извежда и средния му успех.

Програма Zad100.cpp решава задачата. Данните за студент се дефинират чрез структурата:

```
struct student
{int facnom;
 char name[26];
 double marks[NUM];
};
```

където NUM е константа, определяща броя на предметите. Данните за групата са представени чрез масива

```
student table[30];
```

Реализирани са следните процедури и функции:

*void read_student(student&);*

Въвежда стойности на полетата на структура от тип student.

*void print_student(const student &);*

Извежда върху екрана полетата на структура от тип student.

*void sorttable(int n, student[]);*

Сортира компонентите на масив от структури от тип student в низходящ ред по среден успех. Резултатът от сортирането е в същия масив от структури.

*double average(double*);*

Намира средно-аритметичното на елементите на масив от NUM реални числа.

```
// Program Zad100.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
const NUM = 5;
struct student
{int facnom;
 char name[26];
 double marks[NUM];
};
void read_student(student&);
void print_student(const student&);
void sorttable(int n, student[]);
double average(double*);
int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 } while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
   read_student(table[i]);
 cout << "Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(table[i]);
  cout << endl;
 }
 sorttable(n, table);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
```

```
 {print_student(table[i]);
  cout << setw(7) << average(table[i].marks) << endl;
 }
 return 0;
}
void read_student(student& s)
{cout << "fak. nomer: ";
 cin >> s.facnom;
 char p[100];
 cin.getline(p, 100);
 cout << "name: ";
 cin.getline(s.name, 40);
 for (int i = 0; i <= NUM-1; i++)
 {cout << i << " -th mark: ";
  cin >> s.marks[i];
 }
}
void print_student(const student& stud)
{cout << setw(6) << stud.facnom << setw(30) << stud.name;
 for (int i = 0; i <= NUM-1; i++)
 cout << setw(6) << stud.marks[i];
}
void sorttable(int n, student a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i].marks);
  for (int j = i+1; j <= n-1; j++)
    if (average(a[j].marks) > max)
    {max = average(a[j].marks);
     k = j;
    }
  student x = a[i]; a[i] = a[k]; a[k] = x;
 }
}
double average(double* a)
{double s = 0;
```

```
    for (int j = 0; j <= NUM-1; j++)
       s += a[j];
    return s/NUM;
   }
```

Процедурата за сортиране void sorttable(int n, student a[]) е реализирана неефективно тъй като се разместват структури. При структури с големи размери сортирането е много бавно. Реализацията може да се подобри като се създаде масив от **указатели към структурите** – елементи на table. При необходимост от размяна, тя се осъществява не със структурите, а с адресите на съответните им указатели.

## 11.3 Указатели към структури

Дефинират се по общоприетия начин (фиг. 11.4).

**Указател към структура**

<указател_към_структура> ::=

struct <име_на_структура> * <променлива_указател>

[= & <променлива>]$_{опц}$;

където

<променлива> е от тип <име_на_структура>.

Фиг. 11.4 Указател към структура

В C++ запазената дума struct може да се пропусне.

*Пример:*

```
student st1, st2;
...
student *pst = &st1;
...
pst = &st2;
...
```

В резултат за променливата-указател pst се отделят 4В ОП, в които отначало се записва адресът на st1, след което – адресът на st2.

Достъпът до полетата на променлива от тип структура чрез указател към нея се осъществява чрез обръщението:

(*<променлива_указател>).<име_на_поле>

което е еквивалентно на

<променлива_указател> -> <име_на_поле>

За разделител са използвани знаците – и >, записани последователно.

*Пример:* Достъпът до полетата на st2 чрез указателя pst се реализира чрез обръщенията:

```
pst -> facnom
pst -> name
pst -> marks
```

**Задача 101.** Да се модифицира функцията за сортиране sorttable от задача 100, като за сортирането се използва помощен масив от указатели към структурата student.

Пред вид промени и в main ще дадем програмен фрагмент, решаващ задачата. Функциите read_student(), print_student() и average() са пропуснати, тъй като са същите като в задача 100.

```
// Program Zad101.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
const NUM = 5;
struct student
{int facnom;
 char name[26];
 double marks[NUM];
};
void read_student(student&);
void print_student(const student&);
void sorttable(int n, student* []);
double average(double*);
int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
```

```
 student* ptable[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 } while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
 {read_student(table[i]);
  ptable[i] = &table[i];
 }
 cout << "Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(table[i]);
  cout << endl;
 }
 sorttable(n, ptable);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(*ptable[i]);
  cout << setw(7) << average(ptable[i]->marks) << endl;
 }
 return 0;
}
...
void sorttable(int n, student* a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i]->marks);
  for (int j = i+1; j <= n-1; j++)
    if (average(a[j]->marks) > max)
    {max = average(a[j]->marks);
     k = j;
    }
  student* x;
  x = a[i]; a[i] = a[k]; a[k] = x;
```

```
 }
}
...
```

За работа със структури от данни се използва подходът абстракция със структури от данни.

## 11.4 Абстракция със структури от данни

При този подход методите за използване на данните са разделени от методите за тяхното представяне. Програмите се конструират така, че да работят с “абстрактни данни” – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени **конструктори, селектори** и **предикати**, които реализират “абстрактните данни” по конкретен начин.

Ще го илюстрираме чрез следната задача.

**Задача 102.** Да се напише програма, която реализира основните рационално-числови операции – събиране, изваждане, умножение и деление на рационални числа.

Програма Zad102.cpp решава задачата. Тя дефинира функции за събиране, изваждане, умножение и деление на рационални числа като реализира следните общоизвестни правила:

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{n1.d2 + n2.d1}{d1.d2}$$

$$\frac{n1}{d1} - \frac{n2}{d2} = \frac{n1.d2 - n2.d1}{d1.d2}$$

$$\frac{n1}{d1} * \frac{n2}{d2} = \frac{n1.n2}{d1.d2}$$

$$\frac{n1}{d1} / \frac{n2}{d2} = \frac{n1.d2}{d1.n2}.$$

Тези операции лесно могат да се реализират ако има начин за конструиране на рационално число по зададени две цели числа, представящи съответно неговите числител и знаменател и ако има начини, които по дадено рационално число извличат неговите числител и знаменател. Затова в програмата Zad102.cpp са дефинирани функциите:

```
void makerat(rat& r, int a, int b)- която конструира рационално
                                    число r по дадени числител a и
                                    знаменател b;
int numer(rat& r)       - която намира числителя на рационалното
                          число r;
int denom(rat& r)       - която намира знаменателя на рационалното
                           число r,
```

където с rat означаваме типа рационално число.

Все още не знаем как точно да реализираме тези функции, нито как се представя рационално число, но ако ги имаме, функциите за рационално-числова аритметика и процедурата за извеждане на рационално число могат да се реализират по следния начин:

```
rat sumrat(rat& r1, rat& r2)        //събира рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2)+numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat subrat(rat& r1, rat& r2)        // изважда рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2)-numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat multrat(rat& r1, rat& r2)       // умножава рационални числа
{rat r;
 makerat(r, numer(r1)*numer(r2),
            denom(r1)*denom(r2));
 return r;
}
```

```
rat quotrat(rat& r1, rat& r2)       // дели рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2),
            denom(r1)*numer(r2));
 return r;
}
void printrat(rat& r)            // извежда рационално число
{cout << numer(r) << "/" << denom(r) << '\n';
}
```

Сега да се върнем към представянето на рационалните числа, а също към реализацията на примитивните операции: конструктора makerat и селекторите numer и denom. Тъй като рационалните числа са частни на две цели числа, удобно представяне на рационално число е структура от вида:

```
struct rat
{int num, den;
};
```

Тогава примитивните функции, реализиращи конструктора makerat и двата селектора numer и denom, имат вида:

```
void makerat(rat& r, int a, int b)
{r.num = a;
 r.den = b;
}
int numer(rat& r)
{return r.num;
}
int denom(rat& r)
{return r.den;
}
```

Тези функции са включени в Zad102.cpp и са използвани за намиране на сумата, разликата, произведението и делението на рационалните числа ½ и ¾.

```
// Program Zad102.cpp
#include <iostream.h>
struct rat
{int num, den;
```

```
};
void makerat(rat& r, int a, int b)
{r.num = a;
 r.den = b;
}
int numer(rat& r)
{return r.num;
}
int denom(rat& r)
{return r.den;
}
rat sumrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2)+numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat subrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2)-numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat multrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*numer(r2),
            denom(r1)*denom(r2));
 return r;
}
rat quotrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2),
            denom(r1)*numer(r2));
 return r;
}
void printrat(rat& r)
```

```
{cout << numer(r) << "/" << denom(r)<< endl;
}
int main()
{rat r1, r2;
 makerat(r1, 1, 2);
 makerat(r2, 3, 4);
 printrat(sumrat(r1, r2));
 printrat(subrat(r1, r2));
 printrat(multrat(r1, r2));
 printrat(quotrat(r1, r2));
 return 0;
}
```

Реализирането на подхода абстракция със структури от данни в Задача 102, показва следните четири нива на абстракция:

- Използване на рационалните числа в проблемна област (във функцията main);
- Реализиране на правилата за рационално-числова аритметика (sumrat, subrat, multrat, quotrat, printrat);
- Избор на представяне на рационалните числа и реализиране на примитивни конструктори и селектори (makerat, numer и denom);
- Работа на ниво структура.

Използването на подхода прави програмите по-лесни за описание и модификация. Ако разгледаме по-внимателно изпълнението на горната програма, забелязваме, че тя има редица недостатъци, но основният е, че не съкращава рационални числа. За да поправим този недостатък, се налага да променим единствено функцията makerat. За целта ще използваме помощната функция gcd, дефинирана в глава 8. Новата makerat има вида:

```
void makerat(rat& r, int a, int b)
{if (a == 0) {r.num = 0; r.den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a > 0 && b > 0 || a < 0 && b < 0)
  {r.num = abs(a)/g; r.den = abs(b)/g;}
  else {r.num = - abs(a)/g; r.den = abs(b)/g;}
```

```
    }
   }
```

С това проблемът със съкращаването на рационални числа е решен. За разрешаването му се наложи малка модификация на програмата, която засегна само премитивния конструктор makerat. Последното илюстрира лесната модифицируемост на програмите, реализиращи горния подход.

## 11. 5 От структури към класове

В тази задача дефинирахме структурата rat, определяща рационални числа, и реализирахме някой основни функции за работа с такива числа. *Възниква въпросът:* Може ли да използваме тази структура като тип данни рационално число? Отговорът е не, защото при типовете данни представянето на данните е скрито за потребителя. На последния са известни множеството от стойности и операциите и вградените функции, допустими за типа. Така възниква усещането, че трябва да се обедини представянето на рационално число като запис с две полета с примитивните операции (makerat, numer и denom), пряко използващи представянето. Последното е възможно, тъй като в езика C++ се допуска полетата на структура да са функции, разбира се от тип, различен от типа на структурата.

В следващото описание примитивните операции, реализирани чрез функциите makerat, numer и denom, а също функцията за извеждане на рационално число, ще направим полета на структутара rat. За целта реализираме следните две стъпки:

• Включване във фигурните скоби на дефиницията на структурата rat на декларациите:

```
void makerat(rat& r, int a, int b);
int numer(rat& r);
int denom(rat& r);
void printrat(rat& r);
```

от които елеминираме участието на формалния параметър rat& r, тъй като неговата функция ще се изпълни от полетата num и den. Получаваме структурата:

```
struct rat
{int num;
```

```
 int den;
 void makerat(int a, int b);
 int numer();
 int denom();
 void printrat();
};
```

която може да се интерпретира като запис с две полета num и den, над които могат да се изпълняват функциите:

- makerat, която конструира рационалното число a/b чрез свързването на num и den с a и b съответно;
- numer, която намира числителя num на рационалното число num/den;
- denom, която намира знаменателя den на рационалното число num/den;
- printrat, която извежда рационалното число num/den.

• Отразяване на тези промени в дефинициите на функциите. За целта ще изтрием участията на rat& r и r., а между типа и името на всяка функция ще поставим името на структурата rat, следвано от оператора ::.

Получаваме:

```
void rat::makerat(int a, int b)
{if (a == 0) {num = 0; den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a > 0 && b > 0 || a < 0 && b < 0)
  {num = abs(a)/g;
   den = abs(b)/g;
  }
  else
   {num = - abs(a)/g;
    den = abs(b)/g;
   }
 }
}
int rat::numer()
{return num;
```

```
}
int rat::denom()
{return den;
}
void rat::printrat()
{cout << num << "/" << den << endl;
}
```

Тези функции се наричат **член-функции на структурата rat.** Извикването им се осъществява като полета на структура.

*Пример:*

```
rat r;                // дефиниция на променлива от тип rat
r.makerat(1, 5)       // r се свързва с рационалното число 1/5
r.numer()             // намира числителя на r, в случая 1
r.denom()             // връща знаменателя на r, в случая 5
r.printrat()          // извежда върху екрана r.
```

Обръщението r.numer() е еквивалентно на изпълнение на оператора

```
return r.num;
```

Програма Zad102_1.cpp реализира последните промени.

```
Program Zad102_1.cpp
#include <iostream.h>
#include <math.h>
struct rat
{int num;
 int den;
 void makerat(int, int);
 int numer();
 int denom();
 void printrat();
};
void rat::printrat()
{cout << num << "/" << den << endl;
}
int gcd(int a, int b)
```

```
{while (a!=b)
 if (a > b) a = a-b; else b = b-a;
 return a;
}
void rat::makerat(int a, int b)
{if (a == 0) {num = 0; den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a>0 && b>0 || a<0 && b < 0)
  {num = abs(a)/g;
   den = abs(b)/g;
  }
  else
  {num = - abs(a)/g;
   den = abs(b)/g;
  }
 }
}
int rat::numer()
{return num;
}
int rat::denom()
{return den;
}
rat sumrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer() * r2.denom() +
                  r2.numer() * r1.denom(),
                  r1.denom() * r2.denom());
 return r;
}
rat subrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer() * r2.denom() -
                  r2.numer() * r1.denom(),
                  r1.denom() * r2.denom());
 return r;
}
```

```
rat multrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer()*r2.numer(),
                  r1.denom()*r2.denom());
 return r;
}
rat quotrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer()*r2.denom(),
                  r1.denom()*r2.numer());
 return r;
}
int main()
{rat r1; r1.makerat(-1,2);
 rat r2; r2.makerat(3,4);
 sumrat(r1, r2).printrat();
 // или rat r = sumrat(r1, r2); r.print();
 subrat(r1, r2).printrat();
 // или r = subrat(r1, r2); r.printrat();
 multrat(r1, r2).printrat();
 // или r = multrat(r1, r2); r.printrat();
 quotrat(r1, r2).printrat();
 // или r = quotrat(r1, r2); r.printrat();
 return 0;
}
```

Забелязваме, че във функциите sumrat, subrat, multrat и quotrat не се използват полетата на записа num и den, но ако направим опит за използването им даже на ниво main, опитът ще бъде успешен. Последното може да се забрани, ако се използва етикетите private: пред дефиницията на полетата num и den и public: пред декларациите на член-функциите. Структурата rat получава вида:

```
struct rat
{private:
 int num;
 int den;
 public:
 void makerat(int, int);
 int numer();
```

```
    int denom();
    void printrat();
   };
```

Опитът за използването на полетата num и den на структурата rat извън член-функциите вече ще предизвиква грешка.

Етикетите private и public се наричат **спецификатори за достъп**. Всички член-данни, следващи спецификатора за достъп private, са достъпни само за член-функциите от дефиницията на структурата. Всички член-данни и член-функции, следващи спецификатора за достъп public, са достъпни за всяка функция, която е в областта на структурата. Ако специфитаторите за достъп с пропуснати, всички членове са public. Един и същ спецификатор за достъп може да се използва повече от веднъж в една и съща дефиниция на структура.

Така специфицирането на num и den като private прави невъзможно използването им извън член-функциите makerat, numer, denom и printrat.

Ако заменим запазената дума struct със class, последната програма не променя поведението си. Така дефинирахме първия си клас с име rat, създадохме два негови обекта – рационалните числа r1 и r2 и работихме с тях.

**Задача 103.** Като се използва подходът абстракция със структури от данни да се даде ново представяне на задача 100.

```
   // Program Zad103
  #include <iostream.h>
  #include <iomanip.h>
   #include <string.h>
   const NUM = 5;
   class student
   {private:
    int facnom;
    char name[26];
    public:
    double marks[5];
    void read_student();
```

```
 void print_student();
};
void sorttable(int n, student[]);
double average(double*);
int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 }while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
   table[i].read_student();
 cout << "table: \n";
 for (i = 0; i <= n-1; i++)
 {table[i].print_student();
  cout << endl;
 }
 sorttable(n, table);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {table[i].print_student();
  cout << setw(7) << average(table[i].marks) << endl;
}
 return 0;
}
void student::read_student()
{cout << "fak. nomer: ";
 cin >> facnom;
 char p[100];
 cin.getline(p, 100);
 cout << "name: ";
 cin.getline(name, 40);
 for (int i = 0; i <= NUM-1; i++)
```

```
 {cout << i << " -th mark: ";
  cin >> marks[i];
 }
}
void student::print_student()
{cout << setw(6) << facnom << setw(30) << name;
 for (int i = 0; i <= NUM-1; i++)
 cout << setw(6) << marks[i];
}
void sorttable(int n, student a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i].marks);
  for (int j = i+1; j <= n-1; j++)
    if (average(a[j].marks) > max)
    {max = average(a[j].marks);
     k = j;
    }
  student x = a[i]; a[i] = a[k]; a[k] = x;
 }
}
double average(double* a)
{double s = 0;
 for (int j = 0; j <= NUM-1; j++)
    s += a[j];
 return s/NUM;
}
```

Тези идеи са в основата на нов подход за програмиране – обектно – ориентирания.

## Задачи

**Задача 1.** Да се напише функция, която намира разстоянието между две точки в равнината. Като се използва тази функция, да се напише

програма, която въвежда координатите на n точки от равнината, намира и извежда най-голямото разстояние между тях. За целта да се дефинира структура, определяща точка от равнината с координати (x, y).

**Задача 2.** Да се напише програма, която въвежда факултетните номера, имената и успеха по k предмета на студентите от една група и извежда следната таблица:

```
N  име      предмет1    ...   предметK    среден успех
==============================================================
.   .       .            .       .
.   .       .            .       .
.   .       .            .       .
==============================================================
            ср. успех    ...   ср. успех     ср. успех
```

**Задача 3.** Да се напише:

а) булева функция equal(rat x, rat y), която установява дали рационалните числа x и y са равни.

б) булева функция grthen(rat x, rat y), която установява дали рационалното число x е по-голямо от рационалното число y.

в) функция maxrat(int n, rat x[]), която намира най-голямото от рационалните числа на масива x.

г) функция sortrat(int n, rat x[]), която сортира елементите на редицата от рационални числа $x_0$, $x_1$, ..., $x_{n-1}$.

**Задача 4.** Да се напише програма, която решава системата уравнения

$$a\,x + b\,y = e$$
$$c\,x + d\,y = f$$

където коефициентите a, b, c, d, e, f, а също и неизвестните x и y са рационални числа.

**Задача 5.** Нека $a_0$, $a_1$, ..., $a_{n-1}$ и x са рационални числа. Да се напише функция, която намира стойността на полинома

$$P(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n.$$

**Задача 6.** Да се дефинира структура, определяща точка от равнината с координати (x, y), където x и y приемат за стойности числата от 1 до 100. Да се напише програма, която чете координатите на четири точки, представляващи върховете A, B, C и D на четириъгълник в цикличен ред и определя дали ABCD е квадрат, правоъгълник или друга фигура.

**Задача 7.** Да се напише програма, която решава системата уравнения

$$a\,x + b\,y = e$$
$$c\,x + d\,y = f$$

където коефициентите a, b, c, d, e, f, а също и неизвестните x и y са комплексни числа.

**Задача 8.** Нека $a_0$, $a_1$, ..., $a_{n-1}$ и x са комплексни числа. Да се напише функция, която намира стойността на полинома

$$P(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n.$$

**Задача 9.** Дадени са естественото число n и комплексното число z. Да се напише програма, която пресмята стойността на следната комплексна функция:

$$F1 = 1 + \frac{z}{1!} + \frac{z^2}{2!} + \ldots + \frac{z^n}{n!}$$

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. Ал Стивънс, Кл. Уолнъм, C++ библия, АЛЕКС СОФТ, София, 2000.
3. М. Тодорова, Езици за функционално и логическо програмиране – функционално програмиране, СОФТЕХ, София, 1998.
4. Ст. Липман, Езикът C++ в примери, КОЛХИДА ТРЕЙД КООП, София, 1993.
5. Ч. Сфар, Visual C++ 6.0, том 1, СОФТПРЕС, София 2000.

# 12

# Синтезиране на програми на подмножество на езика C++

## 12.1 Необходимост от синтезиране на програми

Нека като част от алголоподобна програма се налага да се намерят частното y1 и остатъкът y2 от целочисленото деление на неотрицателното цяло число x1 на положителното цяло число x2. За решение нека сме написали следния програмен фрагмент:

```
y1=0; y2=x1;
while (y2>x2)
{y1++;
 y2=y2-x2;
}
```

Трябва да проверим дали той наистина решава задачата. За целта най-често се извършва тестване, при което за допустими входни данни се проверява дали се получава правилен резултат. От условието се вижда, че за входни стойности трябва да се изберат такива, които удовлетворяват: $x1 \geq 0 \wedge x2>0$, където x1 и x2 са цели числа, а $\wedge$ означава конюнкция. Когато изпълнението на програмния фрагмент завърши, трябва да е в сила: $x1=y1.x2+y2 \wedge 0 \leq y2<x2$. Условието $y2<x2$ следва от дефиницията за остатък от целочислено деление. Свързваме програмния фрагмент с подходящи коментари, означаващи тези условия, т.е.

```
{φ(x1,x2): x1≥0 ∧ x2>0}
 y1=0; y2=x1;
```

```
 while(y2>x2)
 {y1++;
  y2=y2-x2;
 }
```

{ψ(x1,x2,y1,y2): x1=y1.x2+y2 ∧ 0≤y2<x2}

Предикатът φ(x1,x2) се нарича **предусловие**, а ψ(x1,x2,y1,y2) - **следусловие**. За да извършим тестването, ще оградим програмния фрагмент със следните оператори:

```
cout << "делимо x1= " << x1
     << " делител x2= " << x2 << endl;
```

- пред фрагмента

и

```
cout  << "x1 = y1.x2+y2  " << (x1 == y1*x2+y2) << endl
      << " 0 <= y2 < x2 =  "  << (0 <= y2 && y2 < x2) << endl;
```

- след фрагмента.

При изпълнение на програмния фрагмент за x1==7 и x2==2, се получава:

```
делимо x1 = 7 делител x2 = 2
x1 = y1.x2+y2 = 1
0 <= y2 < x2 = 1,
```

а при изпълнение за x1 == 9 и x2 == 3 -

```
делимо x1 = 9 делител x2 = 3
x1 = y1.x2+y2 = 1
0 <= y2 < x2 = 0.
```

Резултатът не е правилен, тъй като условието y2<x2 не е в сила (y1==2 и y2==3).

След постъпкови изпълнения заключаваме, че ако заменим условието y2>x2 на оператора за цикъл с y2>=x2, изпълнението на програмния фрагмент

{φ(x1,x2): x1≥0 ∧ x2>0}

```
 y1=0; y2=x1;
 while (y2>=x2)
 {y1++;
  y2=y2-x2;
 }
```

{ψ(x1,x2,y1,y2): x1=y1.x2+y2 ∧ 0≤y2<x2}

ще даде правилни резултати в горните два случая. Отново не сме сигурни, дали за всички цели числа x1 и x2, които удовлетворяват предиката $\varphi(x1,x2)$, след завършване на изпълнението на програмния фрагмент, ще е в сила $\psi(x1,x2,y1,y2)$ за текущите стойности на x1, x2, y1 и y2. Ще отбележим също, че особено важно е правилното определяне на предусловието $\varphi$ и следусловието $\psi$.

Но даже и правилно да сме ги определили, след завършване на тестването на програмния фрагмент отново не можем да сме съвсем сигурни, че той е правилен. Можем само да заключим, че фрагментът дава правилни резултати за тези частни случаи, които са използвани при тестването.

За да се убедим, че програма (програмен фрагмент) е правилна относно дадена входно/изходна спецификация, най-добре е с формални средства да извършим доказателство. Такива средства са методите за верификация на програми.

През последните години, доказателството на правилността на програмите заема сравнително важно място в науката информатика. Процесът на програмиране се състои в писане на програми, анотиране на програмите с предусловия и следусловия, удовлетворяващи дадена входно/изходна спецификация и доказване на правилността им чрез различни методи за верификация. Всички популярни методи за верификация на програми обаче са трудоемки. Това води до необходимостта програмите да бъдат синтезирани, т.е. да бъдат строени паралелно с доказателството, че са правилни.

Съществуват различни методи за синтезиране на алголоподобни програми. В настоящата глава ще илюстрираме подхода на преобразуващите предикати, който може да се използва както за синтезиране, така и за верифициране на алголоподобни програми, в частност на програми на езика C++.

## 12.2 Синтезиране на програми на подмножество на C++

За да не усложняваме изложението, ще използваме подмножество на езика, състоящо се от: празен оператор, блок, оператор за

присвояване, условен оператор (пълна и кратка форма) и оператор за цикъл while.

За синтезирането на програми ще използваме специален предикат, който се нарича **преобразуващ предикат**.

**Дефиниция 1.** Нека S е произволен оператор, а R е предикат, който описва очаквания от изпълнението на оператора S резултат. **Преобразуващ предикат** за S и R е предикатът Wp(S, R), представящ множеството от всички състояния, за които изпълнението на S, започващо от такова състояние, завършва и е в сила изходният предикат R.

*Примери:*

а) Нека S е оператора за присвояване a=a+3, а R е предиката a≤3. Тогава Wp(a=a+3, a≤3) = (a ≤ 0), тъй като ако a≤0, изпълнението на a=a+3 завършва и след завършването е в сила a≤3. Освен това, ако е в сила a>0, изпълнението на оператора за присвояване a=a+3 завършва, но не може да направи да е в сила a≤3.

б) Нека S е оператора if (a>=b)c=b; else c=a; а R е предиката c=min(a,b). Изпълнението на оператора S винаги присвоява на c стойността на min(a,b). Следователно Wp(S,R)=T, където с T е означена константата true.

Отначало ще определим преобразуващия предикат за операторите на едно подмножество на езика.

Нека R е произволен предикат.

**Дефиниция 2.** Wp(празен_оператор, R) = R.

**Дефиниция 3.** Wp(x = e, R) = domain(e) ∧ R(x ← e), където domain(e) е предикат, описващ множеството от всички състояния, в които е дефиниран изразът e, а R(x ← e) е предикат, който се получава като в R променливата x се замени с e.

**Пример:** Wp(x=x-2, x=10) = (x-2=10) = (x=12)

**Дефиниция 4.** Wp(S1;S2;...;Sn, R) = Wp(S1,Wp(S2;...;Sn, R)), където S1, S2, ..., Sn са произволни оператори.

**Пример:** Wp(t=x; x=y; y=t, x=x0 ∧ y=y0) =

Wp(t=x, Wp(x=y; y=t, x=x0 ∧ y=y0)) =

$$Wp(t=x,\ Wp(x=y,\ Wp(y=t,\ x=x0 \wedge y=y0))) =$$

$$Wp(t=x,\ Wp(x=y,\ x=x0 \wedge t=y0)) =$$

$$Wp(t=x,\ y=x0 \wedge t=y0) = (y=x0) \wedge (x=y0),$$

където x0 и y0 са произволни начални стойности за променливите x и y.

**Дефиниция 5.** $Wp(\{S1;S2;...;Sn\},\ R) = Wp(S1;S2;...;Sn,\ R)$, където S1, S2,..., Sn са произволни оператори.

**Дефиниция 6.** $Wp(if(B)S1;else\ S2,\ R) = domain(B) \wedge (B \Rightarrow Wp(S1,\ R)) \wedge (\neg B \Rightarrow Wp(S2,\ R))$.

В частност

$$Wp(if(B)S,\ R) = domain(B) \wedge (B \Rightarrow Wp(S,\ R)) \wedge (\neg B \Rightarrow R).$$

Често не е необходимо да се намери преобразуващият предикат $Wp(if(B)S1;else\ S2,\ R)$ или $Wp(if(B)S,\ R)$, а само да се провери дали е в сила импликацията $Q \Rightarrow Wp(if(B)S1;else\ S2,\ R)$ или $Q \Rightarrow Wp(if\ (B)\ S,\ R)$. В тези случаи е полезна следната теорема.

**Теорема 1.** Нека за оператора

if (B) S1; else S2

и предикатите Q и R са в сила условията:

а) $Q \Rightarrow domain(B)$

б) $Q \wedge B \Rightarrow Wp(S1,\ R)$

в) $Q \wedge \neg B \Rightarrow Wp(S2,\ R)$.

Тогава (и само тогава) е в сила

$$Q \Rightarrow Wp(if\ (B)\ S1;\ else\ S2,\ R).$$

Доказателството на тази теорема е дадено в [4].

**Следствие.** Нека за оператора

if (B) S;

и предикатите Q и R са в сила условията:

а) $Q \Rightarrow domain(B)$

б) $Q \wedge B \Rightarrow Wp(S,\ R)$

в) $Q \wedge \neg B \Rightarrow R$.

Тогава (и само тогава) е в сила $Q \Rightarrow Wp(if\ (B)\ S,\ R)$.

**Забележка.** Ако операторът if (B) S1;else S2; или if(B)S; се изпълнява в състояние, при което предикатът B е дефиниран, т.е. domain(B) = T, условие а) от Теорема 1 и следствието отпада.
От Теорема 1 и следствието от нея, ще определим неформално метода на преобразуващите предикати за синтезиране на условен оператор.

**Метод на преобразуващите предикати за**
*а) синтезиране на оператора if/else*

1. Намират се условие B, оператори S1 и S2, така че импликациите:
   $Q \wedge B \Rightarrow Wp(S1, R)$ и
   $Q \wedge \neg B \Rightarrow Wp(S2, R)$ да са в сила.
2. Проверява се дали е в сила $Q \Rightarrow domain(B)$.

*б) синтезиране на оператора if – кратка форма*

1. Намират се условие B и оператор S, така че импликациите:
   $Q \wedge B \Rightarrow Wp(S, R)$ и
   $Q \wedge \neg B \Rightarrow R$ да са в сила.
2. Проверява се дали е в сила $Q \Rightarrow domain(B)$.

Тъй като практически не е лесно да се използва дефиницията на преобразуващия предикат за оператора while, ще формулираме теорема, чрез която може да се провери верността на импликацията

$Q \Rightarrow Wp(while(B)S, R)$.

За целта с оператора за цикъл while(B)S; свързаме:

а) инварианта P – предикат, който е в сила преди изпълнението и след изпълнението на всяка стъпка на оператора while;

б) ограничаваща функция t – целочислена функция, явяваща се горна граница на броя на стъпките на цикъла, които остават да бъдат изпълнени. Функцията t трябва да е ограничена отдолу от 0 и при всяка стъпка от изпълнението на оператора за цикъл да намалява поне с 1.

**Теорема 2.** Нека предикатът P и целочислената функция t удовлетворяват условията:

а) $P \wedge B \Rightarrow Wp(S, P)$

б) $P \wedge B \Rightarrow t > 0$ и

в) $P \wedge B \Rightarrow Wp(t1 = t;\ S,\ t<t1)$,

където t1 е нов идентификатор. Тогава е в сила условието:

$P \Rightarrow Wp(while(B)S,\ P \wedge \neg B)$.

Доказателството на теоремата е дадено в [4].

Като следствие от Теорема 2 може да се формулира списък от условия за верификация и синтез на оператора за цикъл while.

Нека е даден while цикъл, подходящо анотиран с предусловие, инварианта, ограничаваща функция и следусловие:

{Q: предусловие}
{P: инварианта}
{t: ограничаваща функция}
while (B) S;
{R: следусловие}

**Списък от условия за верификация и синтез на оператора за цикъл while**

1) P е в сила преди изпълнението на оператора за цикъл, т.е. или $Q \Rightarrow P$ е в сила, или съществува оператор S0, така че $Q \Rightarrow Wp(S0,\ P)$ е в сила.

2) $P \wedge B \Rightarrow Wp(S,\ P)$ е в сила, т.е. P е инварианта на цикъла.

3) $P \wedge \neg B \Rightarrow R$ е вярно, т.е. в момента на завършване на изпълнението на оператора за цикъл е в сила следусловието.

4) $P \wedge B \Rightarrow t > 0$ е в сила, т.е. t е ограничена отдолу от 0, докато изпълнението на оператора за цикъл не е завършило.

5) $P \wedge B \Rightarrow Wp(t1 = t;\ S,\ t<t1)$ е вярно, т.е. всяка стъпка от изпълнението на оператора за цикъл води до строго намаляване на ограничаващата функция t.

На базата на този списък ще опишем метод за синтезиране на програмни фрагменти, съдържащи оператора за цикъл while.

**Метод на преобразуващите предикати за синтезиране на оператора за цикъл while**

1. Проверява се дали е в сила $Q \Rightarrow P$. Ако това не е така, търси се оператор S0, така че $Q \Rightarrow Wp(S0, P)$ да е в сила.

2. Намира се условие B, така че $P \wedge \neg B \Rightarrow R$ да е в сила.

3. Проверява се дали $P \wedge B \Rightarrow t > 0$ е в сила.

4. Намира се оператор S, така че да са в сила условията:

   $P \wedge B \Rightarrow Wp(t1 = t; S, t<t1)$ и

   $P \wedge B \Rightarrow Wp(S, P)$.

Чрез примери ще илюстрираме метода на преобразуващите предикати за синтезиране на програми на езика C++.

**Задача 104.** Да се синтезира програма на езика C++, която намира частното y1 и остатъка y2 от целочисленото деление на неотрицателното цяло число x1 на положителното цяло число x2.

От условието на задачата, за предусловие и следусловие определяме:

$Q: x1 \geq 0 \wedge x2>0$

$R: x1=y1.x2+y2 \wedge 0 \leq y2<x2$.

Тук y1 и y2 са цели числа, съдържащи съответно частното и остатъка от целочисленото деление x1 на x2.

Тъй като y1 е равно на броя на срещанията на x2 в x1, търсената програма ще съдържа оператор за цикъл. За целта определяме следните инварианта и ограничаваща функция:

$P: x1=y1.x2+y2 \wedge 0 \leq y2 \wedge x2>0$

$t: y2$.

Инвариантата е получена като от R е отстранен конюнктивният член $y2<x2$ и е добавен $x2>0$, съдържащ се в предусловието. Тъй като не е в сила $Q \Rightarrow P$, търсим оператор (редица от оператори) S0, така че $Q \Rightarrow Wp(S0, P)$ да е в сила. Нека

S0: y1=f1(x1,x2); y2=f2(x1,x2,y1).

Имаме:

$Q \Rightarrow Wp(y1=f1(x1,x2); y2=f2(x1,x2,y1), P)$, т.е.

$x1 \geq 0 \wedge x2>0 \Rightarrow$

$x1=f1(x1,x2).x2+f2(x1,x2,f1(x1,x2)) \wedge$

$0 \le f2(x1,x2,f1(x1,x2)) \wedge x2>0$

Ако за f1(x1,x2) и f2(x1,x2,y1) изберем 0 и x1 съответно, импликацията ще бъде в сила. Така получаваме S0: y1 = 0; y2 = x1.

Тъй като P и R са известни, от условие 3, от списъка от условия за верификация и синтез на оператора while, намираме B. Имаме:

$x1=y1.x2+y2 \wedge 0 \le y2 \wedge x2>0 \wedge \neg B \Rightarrow$

$x1=y1.x2+y2 \wedge 0 \le y2<x2,$

т.е. B: $y2 \ge x2$.

Условие 4: $P \wedge B \Rightarrow t > 0$ е в сила, тъй като $y2 \ge x2$ и $x2>0$ са в сила.

Остава да намерим тялото на цикъла. Търсим го от вида:

S: y1=g1(x1,x2,y1,y2);
y2=g2(x1,x2,y1,y2).

Функциите g1(x1,x2,y1,y2) и g2(x1,x2,y1,y2) трябва да са такива, че условия 2 и 5 от списъка от условия за верификация и синтез на оператора while, да са в сила.

От условията:

$P \wedge B \Rightarrow Wp(t1 = t; S, t<t1)$, т.е.

$x1=y1.x2+y2 \wedge 0 \le y2 \wedge x2>0 \wedge y2 \ge x2 \Rightarrow$

$g2(x1,x2,g1(x1,x2,y1,y2),y2) < y2,$

и

$P \wedge B \Rightarrow Wp(S, P)$, т.е.

$x1=y1.x2+y2 \wedge 0 \le y2 \wedge x2>0 \wedge y2 \ge x2 \Rightarrow$

$x1=g1(x1,x2,y1,y2).x2+g2(x1,x2,g1(x1,x2,y1,y2),y2) \wedge$

$0 \le g2(x1,x2,g1(x1,x2,y1,y2), y2) \wedge x2 > 0,$

определяме функциите g1 и g2. Ако за g2(x1,x2,y1,y2) изберем y2-x2, $g2(x1,x2,g1(x1,x2,y1,y2),y2) \ge 0$ ще е в сила. Освен това $g2(x1,x2,g1(x1,x2,y1,y2),y2)<y2$ също е в сила, тъй като $x2>0$. Условието $x1=x2.g1(x1,x2,y1,y2)+g2(x1,x2,g1(x1,x2,y1,y2),y2)$ има вида: $x1=x2.g1(x1,x2,y1,y2)+y2-x2$. Тъй като $x1=x2.y1+y2$ е в сила, за g1(x1,x2,y1,y2) можем да изберем y1+1. Тялото на оператора за цикъл има вида: S: y1++; y2=y2-x2, а синтезираната програма:

```
y1=0; y2=x1;
while (y2>=x2)
{y1++;
y2=y2-x2;
}
```

Съществуват различни подходи за построяване на инварианта на цикъл. Най-често използвани са: *отстраняване на конюнктивен член*, *замяна на константа с променлива*, *комбиниране на предусловие и следусловие*. Ще ги илюстрираме чрез следващите задачи.

**Задача 105.** Дадено е цялото число n, $n \geq 0$. Да се синтезира програма на езика C++, която намира най-голямото цяло число a, чийто квадрат е не по-голям от n.

От условието на задачата, за предусловие и следусловие определяме:

Q: $n \geq 0$

R: $a \geq 0 \wedge a^2 \leq n \wedge n < (a+1)^2$.

*I начин:*

Ако отстраним третия конюнктивен член на R, получаваме инвариантата:

P: $0 \leq a \wedge a^2 \leq n$,

а за ограничаваща функция избираме:

t: $n-a^2$.

От условието "P е в сила преди изпълнението на оператора за цикъл" следва, че трябва да се намери инициализация S0, така че да е в сила $Q \Rightarrow Wp(S0, P)$. Търсим S0 от вида:

a=f1(n);

Имаме:

$n \geq 0 \Rightarrow 0 \leq f1(n) \wedge f1(n)^2 \leq n$.

За да е в сила тази импликация, за f1(n) избираме 0. Получаваме

S0: a=0;

От условието $P \wedge \neg B \Rightarrow R$ ще намерим B. Имаме:

$0 \leq a \wedge a^2 \leq n \wedge \neg B \Rightarrow$

$0 \leq a \wedge a^2 \leq n \wedge n < (a+1)^2$.

Следователно $\neg$ B: $n<(a+1)^2$, а B: $n\geq(a+1)^2$. Забелязваме, че в този случай $\neg$ B е отстранения конюнктивен член.

Условие 4 има вида:

$0\leq a \wedge a^2\leq n \wedge n\geq(a+1)^2 \Rightarrow n-a^2>0$

и следва от $a^2<(a+1)^2\leq n$. Остава да намерим тялото S на оператора за цикъл, така че да са в сила условия 2 и 5 от списъка от условия за верификация и синтез на оператора while. S търсим от вида:

a=f2(n,a);

Тъй като

$Wp(t1=t;\ S,\ t<t1) = a^2<f2^2(n,a)$,

за да е в сила условие 5, за f2(n,a) избираме a+h, където h>0. Условие 2 има вида:

$0\leq a \wedge a^2\leq n \wedge n\geq(a+1)^2 \Rightarrow$

$0\leq a+h \wedge (a+h)^2\leq n$.

Ако за стойност на h изберем 1, това условие ще бъде в сила и синтезираната програма има вида:

```
a=0;
 while (n >= (a+1)*(a+1)) a = a+1;
```

*II начин:*

Ако отстраним втория конюнктивен член на R, получаваме инвариантата:

P: $0\leq a \wedge n<(a+1)^2$,

а за ограничаваща функция избираме:

t: $(a+1)^2-n$

Търсим инициализация

S0: a=f1(n);

така че да е в сила $Q \Rightarrow Wp(S0,P)$. Имаме:

$n\geq 0 \Rightarrow 0\leq f1(n) \wedge n<(f1(n)+1)^2$.

Ако за f1(n) изберем n, горната импликация ще бъде в сила. От условието $P \wedge \neg B \Rightarrow R$, т.е.

$0\leq a \wedge n<(a+1)^2 \wedge \neg B \Rightarrow$

$0\leq a \wedge a^2\leq n \wedge n<(a+1)^2$,

получаваме $\neg$ B: $a^2 \le n$ и B: $n<a^2$. Условие 4 има вида:

$0 \le a \wedge n<(a+1)^2 \wedge n<a^2 \Rightarrow$

$(a+1)^2-n>0$

и очевидно е в сила. Остава да се намери тялото S на оператора за цикъл така, че да са в сила условия 2 и 5 от списъка от условия за верификация и синтез на оператора за цикъл. S търсим от вида:

a=f2(n,a);

Тъй като

$Wp(t1=t;\ S,\ t<t1) = (f2(n,a)+1)^2<(a+1)^2$,

за f2(n,a) можем да изберем a-h, където h>0. При този избор, условие 2 има вида:

$0 \le a \wedge n<(a+1)^2 \wedge n<a^2 \Rightarrow$

$0 \le a-h \wedge n<(a-h+1)^2$.

Ако за h изберем 1, тъй като $n \ge 0$, това условие ще бъде в сила и синтезираната програма има вида:

```
a=n;
 while (n<a*a) a--;
```

*III начин:*

Инвариантата P намираме, като заменим a+1 от R с променливата b и укажем границите на изменение на b, т.е.

P: $a \ge 0 \wedge a^2 \le n<b^2 \wedge a<b \le n+1$,

а за ограничаваща функция избираме:

t: b-a-1.

Търсим инициализация

S0: a=f1(n); b=f2(n,a);

така че да е в сила импликацията Q $\Rightarrow$ Wp(S0,P), т.е.

$n \ge 0 \Rightarrow f1(n) \ge 0 \wedge f1(n)^2 \le n<f2(n,f1(n))^2 \wedge f1(n)<f2(n,f1(n)) \le n+1$.

За f1(n) и f2(n,a) избираме 0 и n+1, съответно. Получаваме:

S0: a=0; b=n+1;

От условието P $\wedge \neg$ B $\Rightarrow$ R, т.е.

$a \ge 0 \wedge a^2 \le n<b^2 \wedge a<b \le n+1 \wedge \neg B \Rightarrow$

$a^2 \le n<(a+1)^2$

следва, B: $b \neq a+1$. Условието $P \wedge B \Rightarrow t>0$, т.е.

$$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \Rightarrow b-a-1>0$$

е в сила, тъй като от $a<b$ следва $b-a-1 \geq 0$, а освен това $b \neq a+1$. Тялото на цикъла търсим от вида:

S: a=g1(n,a,b); b=g2(n,a,b).

Тъй като

Wp(t1=b-a-1; S, b-a-1<t1) =

g2(n,g1(n,a,b), b)-g1(n,a,b)-1<b-a-1,

условие 5 има вида:

$$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \Rightarrow$$

g2(n,g1(n,a,b),b)-g1(n,a,b) < b-a

и някои възможности за да е в сила са:

. g1(n,a,b)=a+1; g2(n,a,b)=b и следователно

S1: a++;

. g1(n,a,b)=a; g2(n,a,b)=b-1 и следователно

S2: b--;

. g1(n,a,b)=(a+b)/2; g2(n,a,b)=b и следователно

S3: a=(a+b)/2;

. g1(n,a,b)=a; g2(n,a,b)=(a+b)/2 и следователно

S4: b=(a+b)/2;

където делението е целочислено. Тъй като $a<b$ и $b \neq a+1$ са в сила, в сила са и $a<(a+b)/2<b$.

За оператора S1, условие 2 има вида:

$$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \Rightarrow$$

$$a+1 \geq 0 \wedge (a+1)^2 \leq n < b^2 \wedge a+1 < b \leq n+1.$$

То не е вярно, но

$$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \wedge (a+1)^2 \leq n \Rightarrow$$

$$a+1 \geq 0 \wedge (a+1)^2 \leq n < b^2 \wedge a+1 < b \leq n+1,$$

т.е.

$$P \wedge B \wedge (a+1)^2 \leq n \Rightarrow Wp(a{+}{+}, P)$$

е в сила, тъй като от $a<b$ следва $a \leq b-1$. Но $b-1 \neq a$ и следователно $a<b-1$.

Аналогично, за оператора S2, условие 2 има вида:

$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \Rightarrow$

$a \geq 0 \wedge a^2 \leq n < (b-1)^2 \wedge a < b-1 \leq n+1.$

То също не е вярно, но

$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \wedge (a+1)^2 > n \Rightarrow$

$a \geq 0 \wedge a^2 \leq n < (b-1)^2 \wedge a < b-1 \leq n+1,$

т.е.

$P \wedge B \wedge (a+1)^2 > n \Rightarrow Wp(b--, P)$

е в сила, тъй като от $a<b$ и $b \neq a+1$ следва $a<b-1$, т.е. $a+1 \leq b-1$. Оттук и от $n<(a+1)^2$ следва $n<(b-1)^2$.

Получихме:

$P \wedge B \wedge (a+1)^2 \leq n \Rightarrow Wp(a++, P)$

$P \wedge B \wedge (a+1)^2 > n \Rightarrow Wp(b--, P)$

От Теорема 1 следва, че е в сила импликацията:

$P \wedge B \Rightarrow Wp(S, P),$

където

S: if ((a+1)*(a+1) <= n) a++; else b--;

Така синтезираната програма има вида:

```
a=0; b=n+1;
while (b!=a+1)
if ((a+1)*(a+1)<=n) a++;
 else b--;
```

За оператора S2, условие 2 ще е в сила също, ако вместо предиката $(a+1)^2>n$ в лявата страна на импликацията се добави конюнктивният член $n<(b-1)^2$, т.е.

$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \wedge n < (b-1)^2 \Rightarrow$

$a \geq 0 \wedge a^2 \leq n < (b-1)^2 \wedge a < b-1 \leq n+1$

е в сила.

Освен това, тъй като и импликацията:

$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \wedge (b-1)^2 \leq n \Rightarrow$

$a+1 \geq 0 \wedge (a+1)^2 \leq n < b^2 \wedge a+1 < b \leq n+1,$

е в сила заради $(a+1)^2 \leq (b-1)^2 \leq n$, от Теорема 1 следва, че е в сила импликацията:

$P \wedge B \Rightarrow Wp(S, P)$,

където

S: if(n<(b-1)*(b-1))b--; else a ++;

Така синтезирахме още една програма, която удовлетворява дадената входно/изходна спецификация:

```
a=0; b=n+1;
while (b!=a+1)
if (n<(b-1)*(b-1)) b--;
else a++;
```

За оператора S3, условие 2 има вида:

$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \Rightarrow$

$(a+b)/2 \geq 0 \wedge ((a+b)/2)^2 \leq n < b^2 \wedge (a+b)/2 < b \leq n+1$

и не е вярно. В сила е обаче импликацията:

$P \wedge B \wedge ((a+b)/2)^2 \leq n \Rightarrow Wp(S3, P)$.

За оператора S4, условие 2 има вида:

$a \geq 0 \wedge a^2 \leq n < b^2 \wedge a < b \leq n+1 \wedge b \neq a+1 \Rightarrow$

$a \geq 0 \wedge a^2 \leq n < ((a+b)/2)^2 \wedge a < (a+b)/2 \leq n+1$

и също не е вярно, но е в сила:

$P \wedge B \wedge n < ((a+b)/2)^2 \Rightarrow Wp(S4, P)$.

Като използваме отново Теорема 1, за тяло на цикъла получаваме:

S: if ((a+b)/2*(a+b)/2<=n) a=(a+b)/2;

else b=(a+b)/2;

а синтезираната програма има вида:

```
a=0; b=n+1;
while (a+1!=b)
if (((a+b)/2)*((a+b)/2)<=n) a=(a+b)/2;
else b=(a+b)/2;
```

която може да се опрости до:

```
a=0; b=n+1;
while (a+1!=b)
{int x=(a+b)/2;
 if (x*x<=n) a=x;
```

```
else b=x;
}
```

Получихме пет програми, които удовлетворяват дадената входно/изходна спецификация. Те са еквивалентни, тъй като са “извлечени” от една и съща входно/изходна спецификация.

**Задача 106.** Да се синтезира програма, която по дадено естествено число n, n≥1, намира n - тото число на Фибоначи.

Анализирайки условието на задачата, за предусловие и следусловие избираме:

Q: $n \geq 1$

R: a=fib(n),

където a е n-тото число на Фибоначи.

Като използваме дефиницията на числата на Фибоначи, за инвариантата и ограничаваща функция определяме:

P: $a=fib(i) \wedge b=fib(i-1) \wedge 1 \leq i \leq n$

t: n-i.

Изборът на P означава, че в произволен момент на изчисление a съдържа i-тото, b съдържа i-1-вото число на Фибоначи, а n-i е горната граница на броя на стъпките на цикъла, които остава да бъдат изпълнени.

От условието "P е в сила преди изпълнението на оператора за цикъл", следва че трябва да построим оператор S0, така че да е в сила импликацията $Q \Rightarrow Wp(S0, P)$. S0 търсим от вида:

```
a=f(n);
b=g(n,a);
i=h(n,a,b);
```

Имаме:

$$n \geq 1 \Rightarrow f(n) = fib(h(n,f(n),g(n,f(n)))) \wedge$$
$$g(n,a) = fib(h(n,f(n),g(n,f(n)))-1) \wedge$$
$$1 \leq h(n,f(n),g(n,f(n))) \leq n.$$

Тази импликация ще е в сила, ако направим следния избор:

$f(n) = 1$, $g(n,a) = 0$ и $h(n,a,b) = 1$.

Така инициализацията получава вида:

S0: a=1; b=0; i=1;

От условието $P \wedge \neg B \Rightarrow R$ ще намерим B. Имаме:

$a=fib(i) \wedge b=fib(i-1) \wedge 1 \le i \le n \wedge \neg B \Rightarrow$

$a=fib(n)$.

Следователно $\neg B$: $i=n$, а $B$: $i \ne n$. Условие 4, от списъка от условия за синтезиране на оператора за цикъл, следва от $i \le n$ и $i \ne n$.

Тялото S на цикъла търсим от вида:

a=f1(n,a,b,i);

b=g1(n,a,b,i);

i=h1(n,a,b,i);

така че да са в сила условия 2 и 5 от списъка от условия за синтезиране на оператора за цикъл.

Тъй като

Wp(t1=t; S, t<t1) =

Wp(t1=n-i; a=f1(n,a,b,i); b=g1(n,a,b,i); i=h1(n,a,b,i), n-i<t1) =

i<h1(n,f1(n,a,b,i),g1(n, f1(n,a,b,i),b,i),i)

условие 5 ще е в сила, ако за h1 изберем i+h, където h е константа и h>0. При този избор, условие 2 има вида:

$a=fib(i) \wedge b = fib(i-1) \wedge 1 \le i \le n \wedge i \ne n \Rightarrow$

$f1(n,a,b,i)=fib(i+h) \wedge$

$g1(n,f1(n,a,b,i),b,i)=fib(i+h-1) \wedge 1 \le i+h \le n$.

От $i \le n$ и $i \ne n$, следва че за h можем да изберем 1. Тогава, според дефиницията на функцията fib, за f1(n,a,b,i) и g1(n,f1(n,a,b,i),b,i) избираме a+b и a, съответно. Получаваме:

f1(n,a,b,i) = a+b;

g1(n,a+b,b,i) = a.

След смяна на променливите имаме: g1(n,a,b,i) = a-b.

Така синтезирахме програмния фрагмент:

```
a=1; b=0; i=1;
while (i!=n)
{a=a+b;
 b=a-b;
 i++;
}
```

където a съдържа n-тото число на Фибоначи.

**Задача 107.** Дадена е числовата редица: $x_0$, $x_1$, ..., $x_{n-1}$, $n \geq 2$. Да се синтезира програма, която установява дали редицата е монотонно растяща.

От условието на задачата, за предусловие и следусловие избираме:

Q: $n \geq 2$

R: $z=(\forall k: 0 \leq k \leq n-2: x_k \leq x_{k+1})$.

I начин:

Ако в R заменим n с i и укажем как i се променя, за инварианта получаваме:

P: $z=(\forall k: 0 \leq k \leq i-2: x_k \leq x_{k+1}) \wedge 2 \leq i \leq n$,

а за ограничаваща функция избираме:

t: $n-i$,

означаваща броя на останалите за сканиране елементи на редицата.

От условието "P е в сила преди изпълнението на оператора за цикъл" следва, че трябва да се намери инициализация S0, така че да е в сила импликацията $Q \Rightarrow Wp(S0, P)$. S0 търсим от вида:

i=f1(n);

z=f2(n,i).

Имаме:

$n \geq 2 \Rightarrow f2(n,f1(n)) = (\forall k: 0 \leq k \leq f1(n)-2: x_k \leq x_{k+1}) \wedge 2 \leq f1(n) \leq n$.

За да е в сила тази импликация, за f1(n) и f2(n,i) избираме 2 и $x_0 \leq x_1$, съответно. Получаваме

S0: i=2; z=x[0]<=x[1];

От условието $P \wedge \neg B \Rightarrow R$, т.е.

$z=(\forall k: 0 \leq k \leq i-2: x_k \leq x_{k+1}) \wedge 2 \leq i \leq n \wedge \neg B \Rightarrow$

$z = (\forall k: 0 \leq k \leq n-2: x_k \leq x_{k+1})$

намираме $\neg$ B: $i=n \vee \neg z$ и оттук B: $(i \neq n) \wedge z$. Условие 4 има вида:

$z=(\forall k: 0 \leq k \leq i-2: x_k \leq x_{k+1}) \wedge 2 \leq i \leq n \wedge (i \neq n) \wedge z \Rightarrow$

$n-i>0$

и е в сила. Остава да намерим тялото S на цикъла, така че да са в сила условия 2 и 5 от списъка от условия за верификация и синтез на оператора while. S търсим от вида:

i=g1(n,i,z);

z=g2(n,i,z);

Тъй като

Wp(t1=t; S, t<t1) = n-g1(n,i,z) < n-i,

за да е в сила условие 5, за g1(n,i,z) избираме i+h, където h>0. Условие 2 има вида:

$z=(\forall k: 0\le k\le i-2: x_k\le x_{k+1}) \wedge 2\le i\le n \wedge (i\ne n) \wedge z \Rightarrow$

$g2(n,i+h,z)=(\forall k: 0\le k\le i+h-2: x_k\le x_{k+1}) \wedge 2\le i+h\le n$

Ако за стойности на h и g2(n,i+h,z) изберем 1 и $z\wedge(x_{i-1}\le x_i)$, съответно, това условие ще бъде в сила. Тъй като z се съдържа в лявата страна на импликацията, т.е. е true, g2(n,i+1,z) опростяваме до $x_{i-1}\le x_i$. Тогава $g2(n,i,z) = x_{i-2}\le x_{i-1}$ и тялото S на оператора за цикъл има вида:

i++;

z=x[i-2]<=x[i-1];

а синтезираният програмен фрагмент:

```
int i=2;
bool z=x[0]<=x[1];
while (i!=n && z)
{i++;
 z=x[i-2]<=x[i-1];
 }
```

където z съдържа резултата.

II начин:

Ако в R заменим 0 с i и укажем как i се променя, за инварианта получаваме:

$P: z=(\forall k: i\le k\le n-2: x_k\le x_{k+1}) \wedge 0\le i\le n-2,$

а за ограничаваща функция избираме:

t: i.

Търсим редица от оператори S0 от вида:

i=f1(n);

z=f2(n,i);

така че да е в сила импликацията Q $\Rightarrow$ Wp(S0, P). Имаме:

$n \geq 2 \Rightarrow f2(n, f1(n)) = (\forall k: f1(n) \leq k \leq n-2: x_k \leq x_{k+1}) \wedge 0 \leq f1(n) \leq n-2$

За да е в сила тази импликация, за f1(n) избираме n-2, а за f2(n, f1(n)) – $x_{n-2} \leq x_{n-1}$. Получаваме

```
S0: i=n-2;
    z=x[n-2]<=x[n-1];
```

От условието P $\wedge \neg$ B $\Rightarrow$ R, т.е.

$z=(\forall k: i \leq k \leq n-2: x_k \leq x_{k+1}) \wedge 0 \leq i \leq n-2 \wedge \neg B \Rightarrow$

$z=(\forall k: 0 \leq k \leq n-2: x_k \leq x_{k+1})$

намираме $\neg$ B: $(i=0) \vee \neg z$, а оттук B: $(i \neq 0) \wedge z$. Условие 4 има вида:

$z=(\forall k: i \leq k \leq n-2: x_k \leq x_{k+1}) \wedge 0 \leq i \leq n-2 \wedge (i \neq 0) \wedge z \Rightarrow$

$i>0$

и очевидно е в сила. Остава да намерим тялото S на оператора за цикъл, така че да са в сила условия 2 и 5 от списъка от условия за синтез на оператора while. S търсим от вида:

```
i=g1(n,i,z);
z=g2(n,i,z);
```

Тъй като

Wp(t1=t; S, t<t1) = g1(n,i,z)<i,

за да е в сила условие 5, за g1(n,i,z) избираме i-h, където h>0. Условие 2 има вида:

$z=(\forall k: i \leq k \leq n-2: x_k \leq x_{k+1}) \wedge 0 \leq i \leq n-2 \wedge (i \neq 0) \wedge z \Rightarrow$

$g2(n,i-h,z)=(\forall k: i-h \leq k \leq n-2: x_k \leq x_{k+1}) \wedge 0 \leq i-h \leq n-2$

Ако за стойности на h и g2(n,i-h,z) изберем 1 и $z \wedge (x_{i-1} \leq x_i)$, съответно, тази импликация ще бъде в сила. Тъй като z участва в лявата част на импликацията и следователно е в сила, g2(n,i-1,z) опростяваме $x_{i-1} \leq x_i$, а оттук g2(n,i,z) = $x_i \leq x_{i+1}$. Следователно, тялото S на оператора за цикъл има вида:

```
i--;
z=x[i-1]<=x[i];
```

а синтезираният програмен фрагмент –

```
int i=n-2;
bool z=x[n-2]<=x[n-1];
```

```
while (i!=0 && z)
{i--;
 z=x[i]<=x[i+1];
}
```

където z съдържа резултата.

**Задача 108.** Дадена е числовата редица: $x_0$, $x_1$, ..., $x_{n-1}$, $n \geq 2$. Да се синтезира програма, която установява дали редицата се състои от различни елементи.

От условието на задачата, за предусловие и следусловие определяме:

Q: $n \geq 2$

R: $z = (\forall k: 0 \leq k \leq n-2: (\forall l: k+1 \leq l \leq n-1: x_k \neq x_l))$.

За инварианта избираме:

P: $z=((\forall k: 0 \leq k \leq i-1: (\forall l: k+1 \leq l \leq n-1: x_k \neq x_l)) \wedge (\forall l: i+1 \leq l \leq j: x_i \neq x_l))$

$\wedge\ 0 \leq i \leq n-2 \wedge i \leq j \leq n-1$

а за ограничаваща функция –

$$t: \frac{(n-i-1).(n-i)}{2} + n - j$$

означаваща броя на останалите двойки, за които трябва да се провери, дали са различни. Търсим инициализация S0 от вида:

```
i=f1(n);
j=f2(n,i);
z=f3(n,i,j);
```

така че импликацията $Q \Rightarrow Wp(S0, P)$ да е в сила. Имаме:

$n \geq 2 \Rightarrow$

$f3(n,f1(n),f2(n,f1(n)))=((\forall k: 0 \leq k \leq f1(n)-1: (\forall l: k+1 \leq l \leq n-1: x_k \neq x_l))$

$\wedge (\forall l: f1(n)+1 \leq l \leq f2(n,f1(n)): x_{f1(n)} \neq x_l)) \wedge$

$0 \leq f1(n) \leq n-2 \wedge f1(n) \leq f2(n,f1(n)) \leq n-1$.

За да е в сила тази импликация, за f1(n), f2(n,i) и f3(n,i,j) избираме 0, 1 и $x_0 \neq x_1$, съответно. Така получаваме

S0: i=0;
j=1;
z=x[0]!=x[1];

От условието $P \wedge \neg B \Rightarrow R$, т.е.

$z=((\forall k: 0\le k\le i-1: (\forall l: k+1\le l\le n-1: x_k\ne x_l)) \wedge (\forall l: i+1\le l\le j: x_i\ne x_l))$

$\wedge\ 0\le i\le n-2 \wedge i\le j\le n-1 \wedge \neg B \Rightarrow$

$z = (\forall k: 0\le k\le n-2: (\forall l: k+1\le l\le n-1: x_k \ne x_l))$

намираме $\neg B$: $(i=n-2 \wedge j=n-1) \vee \neg z$, а оттук $B$: $(i\ne n-2 \vee j\ne n-1) \wedge z$.

Условие 4 има вида:

$z=((\forall k: 0\le k\le i-1: (\forall l: k+1\le l\le n-1: x_k\ne x_l)) \wedge (\forall l: i+1\le l\le j: x_i\ne x_l))$

$\wedge\ 0\le i\le n-2 \wedge i\le j\le n-1 \wedge (i\ne n-2 \vee j\ne n-1) \wedge z \Rightarrow$

$$\frac{(n-i-1).(n-i)}{2}+n-j > 0$$

и е в сила (поотделно се разглеждат двата случая $i\ne n-2$ и $j\ne n-1$).

Остава да намерим тялото на цикъла S, така че да са в сила условия 2 и 5 от списъка от условия за верификация и синтез на оператора while.

S търсим от вида:

i=g1(n,i,j,z);
j=g2(n,i,j,z);
z=g3(n,i,j,z);

Тъй като

Wp(t1=t; S, t<t1) =

$$\frac{(n-g1(n,i,j,z)-1).(n-g1(n,i,j,z))}{2}+n-g2(n,g1(n,i,j,z),j,z) <$$

$$\frac{(n-i-1).(n-i)}{2}+n-j$$

за да е в сила условие 5, някои възможности са:

a) g1(n,i,j,z) = i, g2(n,g1(n,i,j,z),j,z) = j+1, т.е.
g2(n,i,j,z) = j+1

В този случай условие 2 има вида:

$z=((\forall k: 0\le k\le i-1: (\forall l: k+1\le l\le n-1: x_k\ne x_l)) \wedge (\forall l: i+1\le l\le j: x_i\ne x_l))$

$\wedge\ 0\le i\le n-2 \wedge i\le j\le n-1 \wedge (i\ne n-2 \vee j\ne n-1) \wedge z \Rightarrow$

$g3(n,i,j+1,z)=((\forall k: 0\le k\le i-1: (\forall l: k+1\le l\le n-1: x_k\ne x_l)) \wedge$

$(\forall l: i+1 \le l \le j+1: x_i \ne x_l)) \wedge 0 \le i \le n-2 \wedge i \le j+1 \le n-1$

То не е в сила, но ако добавим в лявата страна на импликацията предиката $j \ne n-1$ и за $g3(n,i,j+1,z)$ и изберем $z \wedge (x_i \ne x_{j+1})$, ще е в сила. Тъй като z се съдържа в лявата страна на импликацията по-горе, булевата функция $g3(n,i,j+1,z)$ можем да опростим до $x_i \ne x_{j+1}$. Тогава $g3(n,i,j,z) = x_i \ne x_j$. Следователно,

$P \wedge B \wedge j \ne n-1 \Rightarrow Wp(S1, P)$

е в сила, където

S1: j++; z=x[i]!=x[j];

б) $g1(n,i,j,z) = i+1$, $g2(n,g1(n,i,j,z),j,z) = g2(n,i+1,j,z) = i+2$, т.е. $g2(n,i,j,z) = i+1$

В този случай условие 2 има вида:

$z=((\forall k: 0 \le k \le i-1: (\forall l: k+1 \le l \le n-1: x_k \ne x_l)) \wedge (\forall l: i+1 \le l \le j: x_i \ne x_l))$

$\wedge\ 0 \le i \le n-2 \wedge i \le j \le n-1 \wedge (i \ne n-2 \vee j \ne n-1) \wedge z \Rightarrow$

$g3(n,i+1,i+2,z)=((\forall k: 0 \le k \le i: (\forall l: k+1 \le l \le n-1: x_k \ne x_l)) \wedge$

$(\forall l: i+2 \le l \le i+2: x_{i+1} \ne x_l)) \wedge 0 \le i+1 \le n-2 \wedge i+1 \le i+2 \le n-1$

То не е в сила, но ако добавим в лявата страна на импликацията условието $j=n-1$, тя ще е в сила, ако за $g3(n,i+1,i+2,z)$ изберем $z \wedge (x_{i+1} \ne x_{i+2})$ /Лявата страна на импликацията след опростяване в този случай получава вида $P \wedge i \ne n-2 \wedge j=n-1 \wedge z$, което доказва и неравенството $i+2 \le n-1$/. Тъй като z се съдържа в лявата страна на импликацията, $g3(n,i+1,i+2,z)$ опростяваме до $x_{i+1} \ne x_{i+2}$, а оттук $g3(n,i,j,z) = x_i \ne x_j$. Следователно,

$P \wedge B \wedge j=n-1 \Rightarrow Wp(S2, P)$

е в сила, където

S2: i++; j=i+1; z=x[i]!=x[j].

От метода на преобразуващите предикати за синтезиране на оператора if/else, следва верността на импликацията:

$P \wedge B \Rightarrow Wp(S, P)$,

където

```
S: if (j!=n-1)
   {j++;
```

```
 z=x[i]!=x[j];
}
else
{i++;
 j=i+1;
 z=x[i]!=x[j];
}
```

Синтезираният програмен фрагмент има вида:

```
i=0; j=1; z=x[0]!=x[1];
while ((i!=n-2||j!=n-1) && z)
if (j!=n-1)
{j++;
 z=x[i]!=x[j];
}
else
{i++;
 j=i+1;
 z=x[i]!=x[j];
}
```

където z съдържа резултата.

**Задача 109.** Дадена е сортираната във възходящ ред редица от числа $b_0$, $b_1$, ..., $b_{n-1}$, $n>1$. Да се синтезира програма, която по дадено число x, така че $b_0 \le x < b_{n-1}$, намира къде в масива b числото x може да се вмъкне, така че да се запази наредбата.

За предусловие и следусловие избираме:

Q: $n>1 \wedge sort(b, 0..n-1) \wedge b_0 \le x < b_{n-1}$

R: $0 \le i < n-1 \wedge b_i \le x < b_{i+1}$,

където параметърът i съдържа резултата, а със sort(b, 0..n-1) е означен предикатът $b_0 \le b_1 \le ... \le b_{n-1}$.

За инварианта и ограничаваща функция определяме:

P: $0 \le i < j \le n-1 \wedge b_i \le x < b_j \wedge sort(b, 0..n-1)$ и

t: j-i.

Инвариантата е получена, като в следусловието R, i+1 е заменено с j и е указано как j се променя, а също е добавен конюнктивен член от предусловието.

От условието "Q е в сила преди изпълнението на оператора за цикъл" следва, че трябва да се намери оператор S0, така че Q ⇒ Wp(S0, P) да е в сила. S0 търсим от вида:

i=f(n);

j=g(n,i);

Имаме:

Q ⇒ Wp(i = f(n); j = g(n,i), P), т.е.

$n>1 \wedge sort(b, 0..n-1) \wedge b_0 \le x < b_{n-1} \Rightarrow$

$0 \le f(n) < g(n,f(n)) \le n-1 \wedge b_{f(n)} \le x < b_{g(n,f(n))} \wedge sort(b, 0..n-1)$

Ако за f(n) и g(n,f(n)) изберем 0 и n-1 съответно, тази импликация ще е в сила и за S0 получаваме

i=0;

j=n-1;

От условието:

$P \wedge \neg B \Rightarrow R$, т.е.

$0 \le i < j \le n-1 \wedge b_i \le x < b_j \wedge sort(b, 0..n-1) \wedge \neg B \Rightarrow$

$0 \le i < n-1 \wedge b_i \le x < b_{i+1}$,

за B избираме $j \ne i+1$. Тъй като импликацията $P \wedge B \Rightarrow t > 0$ е в сила, ограничаващата функция t е добре избрана. Остава да намерим тялото на цикъла S, така че да са в сила условия 2 и 5 от списъка от условия за синтезиране на оператора за цикъл. S търсим от вида:

i=f1(n,i,j);

j=g1(n,i,j);

От условието:

P ∧ B ⇒ Wp(t1= t; S, t<t1), т.е.

$0 \le i < j \le n-1 \wedge b_i \le x < b_j \wedge sort(b, 0..n-1) \wedge j \ne i+1 \Rightarrow$

g1(n,f1(n,i,j),j)−f1(n,i,j) < j-i

определяме някои възможности за f1 и g1:

| | |
|---|---|
| f1(n,i,j) = i+1 | f1(n,i,j) = i |
| g1(n,f1(n,i,j),j) = j | g1(n,f1(n,i,j),j) = j-1 |

$f1(i,n,j) = (i+j)/2$ $\qquad$ $f1(n,i,j) = i$

$g1(n,f1(n,i,j),j) = j$ $\qquad$ $g1(n,f1(n,i,j),j) = (i+j)/2,$

За тяло на цикъла получаваме:

S1: i++; $\qquad$ S2: j--;;

S3: i=(i+j)/2 $\qquad$ S4: j=(i+j)/2;

За оператора S1, условие 2 има вида:

$0 \le i < j \le n-1 \wedge b_i \le x < b_j \wedge sort(b, 0..n-1) \wedge j \ne i+1 \Rightarrow$

$0 \le i+1 < j \le n-1 \wedge b_{i+1} \le x < b_j \wedge sort(b, 0..n-1)$

Тази импликация не е в сила, но единственото, което й пречи е предикатът $b_{i+1} \le x$. Ако го добавим като конюнктивен член в лявата страна на импликацията, е в сила

$P \wedge B \wedge b_{i+1} \le x \Rightarrow Wp(S1, P).$

За оператора S2, условие 2 има вида:

$0 \le i < j \le n-1 \wedge b_i \le x < b_j \wedge sort(b, 0..n-1) \wedge j \ne i+1 \Rightarrow$

$0 \le i < j-1 \le n-1 \wedge b_i \le x < b_{j-1} \wedge sort(b, 0..n-1)$

Тази импликация също не е в сила, но ако влючим в лявата страна на импликацията конюнктивния член $x < b_{i+1}$тя ще стане в сила, т.е. ще е в сила:

$P \wedge B \wedge x < b_{i+1} \Rightarrow Wp(S2, P).$

За целта трябва да докажем, че е от лявата страна следва $x < b_{j-1}$. Тъй като $i < j-1$ е в сила, то $i+1 \le j-1$ и съответно $b_{i+1} \le b_{j-1}$ са в сила. Тъй като $x < b_{i+1}$ е в сила, импликацията е доказана. От S1 и S2 образуваме условния оператор

```
S: if(b[i+1]<=x) i++;
   else j--;
```

Той ще може да е тяло на цикъла, тъй като са в сила условията от Теорема 1.

Така синтезирахме следния програмен фрагмент:

```
i=0;
j=n-1;
while(i+1!=j)
if (b[i+1]≤x) i++;
else j--;
```

който реализира линейно търсене на елемент в масив.

За оператора S3, условие 2, от списъка от условия за синтезиране на оператора за цикъл, има вида:

$0 \le i < j \le n-1 \wedge b_i \le x < b_j \wedge sort(b, 0..n-1) \wedge j \ne i+1 \Rightarrow$

$0 \le (i+j)/2 < j \le n-1 \wedge b_{(i+j)/2} \le x < b_j \wedge sort(b, 0..n-1)$.

То ще е в сила, ако в лявата страна на импликацията като конюнктивен член се добави предикатът $b_{(i+j)/2} \le x$, т.е. в сила е:

$P \wedge B \wedge b_{(i+j)/2} \le x \Rightarrow Wp(S3, P)$.

За S4, условие 2, от списъка от условия за синтезиране на оператора за цикъл, има вида:

$0 \le i < j \le n-1 \wedge b_i \le x < b_j \wedge sort(b, 0..n-1) \wedge j \ne i+1 \Rightarrow$

$0 \le i < (i+j)/2 \le n-1 \wedge b_i \le x < b_{(i+j)/2} \wedge sort(b, 0..n-1)$.

То ще е в сила, ако в лявата страна на импликацията като конюнктивен член се добави предикатът $x < b_{(i+j)/2}$, т.е. в сила е:

$P \wedge B \wedge x < b_{(i+j)/2} \Rightarrow Wp(S4, P)$.

Тъй като са в сила предпоставките от Теорема 1, следва, че е в сила импликацията $P \wedge B \Rightarrow Wp(S, P)$, където операторът S има вида:

```
if(b[(i+j)/2]<=x) i=(i+j)/2;
else j=(i+j)/2;
```

Така синтезирахме и програмата:

```
i=0; j=n-1;
while (i+1!=j)
if (b[(i+j)/2] <= x) i = (i+j)/2;
else j = (i+j)/2;
```

която може да се опрости до:

```
i=0; j=n-1;
while (i+1!=j)
{int k=(i+j)/2;
 if (b[k] <= x) i=k;
 else j=k;
}
```

Тя реализира двоично търсене на елемент в редица.

**Задача 110.** Дадени са целочислен двумерен масив b[0:m-1][0:n-1], m>0, n>0 и стойност x, която се съдържа в масива b. Да се синтезира програма, която определя мястото на първото срещане на x във b.

От условието на задачата за предусловие и следусловие определяме:

Q: $m>0 \wedge n>0 \wedge x \in b[0:m-1][0:n-1]$

R: $0 \le i<m \wedge 0 \le j<n \wedge x=b[i][j] \wedge x \notin b[0:i-1][0:n-1] \wedge x \notin b[i][0:j-1]$,

където i и j съдържат резултата.

Ако от R отстраним третия конюнктивен член, за инварианта получаваме:

P: $0 \le i<m \wedge 0 \le j<n \wedge x \in b[0:m-1][0:n-1] \wedge$
$x \notin b[0:i-1][0:n-1] \wedge x \notin b[i][0:j-1]$.

За ограничаваща функция избираме:

t: (m-i).n-j,

означаваща броя на елементите на масива b, които остава да бъдат сканирани при търсене на x.

Търсим инициализация S0 от вида:

i=f(m,n);

j=g(m,n,i);

така че да е в сила $Q \Rightarrow Wp(S0, P)$.

Имаме $m>0 \wedge n>0 \wedge x \in b[0:m-1][0:n-1] \Rightarrow$

$0 \le f(m,n)<m \wedge 0 \le g(m,n,f(m,n))<n \wedge x \in b[0:m-1][0:n-1] \wedge$
$x \notin b[0:f(m,n)-1][0:n-1] \wedge x \notin b[f(m,n)][0:g(m,n,f(m,n))-1]$.

Тази импликация ще е в сила, ако и за f(m,n), и за g(m,n,i) изберем 0. Използвахме, че b[0:-1][0:n-1] и b[f(m,n)][0:-1] са празни множества. Следователно, S0 има вида:

S0: i=0; j=0;

От условието:

$P \wedge \neg B \Rightarrow R$, т.е.

$0 \le i < m \wedge 0 \le j < n \wedge x \in b[0:m-1][0:n-1] \wedge$

$x \notin b[0:i-1][0:n-1] \wedge x \notin b[i][0:j-1] \wedge \neg B \Rightarrow$

$0 \le i < m \wedge 0 \le j < n \wedge x = b[i][j] \wedge x \notin b[0:i-1][0:n-1] \wedge x \notin b[i][0:j-1]$

намираме B: $x \ne b[i][j]$.

Условието $P \wedge B \Rightarrow t > 0$ има вида:

$P \wedge B \Rightarrow (m-i).n-j > 0$.

Изразът $(m-i).n-j$ показва колко елемента от масива b остава да бъдат сканирани при търсене на двойка (i,j), така че $x=b[i][j]$. От верността на условието $P \wedge B$ следва, че всички сканирани до момента елементи на b са различни от x. Останалите елементи са $(m-i).n-j-1$ на брой. Тъй като $x \in b[0:m-1][0:n-1]$, $(m-i).n-j-1 > 0$ е в сила, условие 4 също е изпълнено. Формалното доказателство на горното твърдение, предложено от Тр. Трифонов, е приведено по-долу.

От вярността на конюнкцията $P \wedge B$ следва:

$i<m-1 \vee (i=m-1 \wedge j<n-1)$.

а) $i<m-1$

Следователно $m-i>1$ е в сила. Умножаваме с $n>0$ и получаваме $(m-i).n>n$. Но $n>j$ е в сила също, откъдето $t = (m-i).n-j > 0$.

б) $i=m-1 \wedge j<n-1$

В този случай $m-i=1$, а $t = n-j$. От верността на B следва, че $t>0$.

Тялото S на цикъла търсим от вида:

```
i=f1(m,n,i,j);
j=g1(m,n,i,j);
```

така, че да са в сила условия 2 и 5 от списъка от условия за синтезиране на оператора while.

За да е в сила импликацията

$P \wedge B \Rightarrow Wp(t1 = t;\ S,\ t<t1)$, т.е.

$0 \le i < m \wedge 0 \le j < n \wedge x \in b[0:m-1][0:n-1] \wedge$

$x \notin b[0:i-1][0:n-1] \wedge x \notin b[i][0:j-1] \wedge x \ne b[i][j] \Rightarrow$

$(m-f1(m,n,i,j)).n-g1(m,n,f1(m,n,i,j)) < (m-i).n-j$,

две възможности за f1 и g1 са:

$f1(m,n,i,j) = i+1$ и $f1(m,n,i,j) = i$

$g1(m,n,f1(m,n,i,j), j) = j$ $\quad$ $g1(m,n,f1(m,n,i,j),j) = j+1$.

От тях получаваме:

S1: i++ и

S2: j++.

За S1, условие 2 има вида:

$0 \le i < m \wedge 0 \le j < n \wedge x \in b[0:m-1][0:n-1] \wedge x \notin b[0:i-1][0:n-1] \wedge$

$x \notin b[i][0:j-1] \wedge x \ne b[i][j] \Rightarrow$

$0 \le i+1 < m \wedge 0 \le j < n \wedge x \in b[0:m-1][0:n-1] \wedge$

$x \notin b[0:i][0:n-1] \wedge x \notin b[i+1][0:j-1]$

Условието $i+1<m$ ще е в сила ако поискаме да е в сила $i \ne m-1$. Условието $x \notin b[0:i][0:n-1]$ ще е в сила ако поискаме да е в сила $j=n-1$. Условието $x \notin b[i+1][0:j-1]$ ще е в сила, ако разширим S1 с оператора за присвояване j = 0, т.е. ако S1 има вида:

S1: i++; j=0;

В резултат имаме:

$0 \le i < m \wedge 0 \le j < n \wedge x \in b[0:m-1][0:n-1] \wedge x \notin b[0:i-1][0:n-1] \wedge$

$x \notin b[i][0:j-1] \wedge x \ne b[i][j] \wedge i \ne m-1 \wedge j=n-1 \Rightarrow$

$0 \le i+1 < m \wedge 0 \le 0 < n \wedge x \in b[0:m-1][0:n-1] \wedge x \notin b[0:i][0:n-1] \wedge$

$x \notin b[i+1][0:-1]$

Условието $i \ne m-1$ е излишно тъй като ако допуснем, че $i=m-1$ е в сила, от $j=n-1$, ще следва, че $x \notin b[0:m-1, 0:n-1]$, което не е вярно.

Следователно

$P \wedge B \wedge j=n-1 \Rightarrow Wp(S1, P)$

е в сила.

За оператора S2 условие 2 има вида:

$0 \le i < m \wedge 0 \le j < n \wedge x \in b[0:m-1][0:n-1] \wedge x \notin b[0:i-1][0:n-1] \wedge$

$x \notin b[i][0:j-1] \wedge x \ne b[i][j] \Rightarrow$

$0 \le i < m \wedge 0 \le j+1 < n \wedge x \in b[0:m-1][0:n-1] \wedge x \notin b[0:i-1][0:n-1] \wedge$

$x \notin b[i][0:j]$

То ще е в сила ако в лявата страна на импликацията добавим $j \ne n-1$, т.е.

$P \wedge B \wedge j \ne n-1 \Rightarrow Wp(S2, P)$

Тъй като са в сила предпоставките на Теорема 1, получаваме

$P \wedge B \Rightarrow Wp(S, P)$

където операторът S има вида

```
if (j==n-1) {i++; j=0;}
else j++;
```

Така синтезирахме програмия фрагмент:

```
i=0; j=0;
while(x!=b[i][j])
if(j==n-1) {i++; j=0;}
else j++;
```

където (i,j) е позицията на първото срещане на елемента x в масива b.

## Задачи

**Задача 1.** Да се синтезира програмен фрагмент, който проверява дали дадено естествено число е просто.

**Задача 2.** Да се синтезира програмен фрагмент, който намира максималния елемент на едномерния масив от числа $x_0$, $x_1$, ..., $x_{n-1}$, $n \geq 1$.

**Задача 3.** Да се синтезира програмен фрагмент, който намира сумата от положителните и броя на отрицателните елементи на едномерния масив от числа $x_0$, $x_1$, ..., $x_{n-1}$, $n \geq 1$.

**Задача 4.** Да се синтезира програмен фрагмент, който проверява дали числото x принадлежи на едномерния масив от числа $a_0$, $a_1$, ..., $a_{n-1}$, $n \geq 1$.

**Задача 5.** Да се синтезира програмен фрагмент, който сортира едномерния масив от числа $x_0$, $x_1$, ..., $x_{n-1}$, $n \geq 1$.

**Задача 6.** Да се синтезира програмен фрагмент, който слива сортираните във възходящ ред едномерни масиви от числа $x_0$, $x_1$,..., $x_{n-1}$ и $y_0$, $y_1$,..., $y_{m-1}$, където $n \geq 1$ и $m \geq 1$.

**Задача 7.** Да се синтезира програмен фрагмент, който проверява дали числото x принадлежи на матрицата A[nxn], $n \geq 1$.

**Задача 8.** Да се синтезира програмен фрагмент, който проверява дали матрицата A[nxn], n≥1 е симетрична относно главния диагонал.

**Задача 9.** Да се синтезира програмен фрагмент, който проверява дали числото x се съдържа под главния диагонал на матрицата A[nxn], n≥1.

**Задача 10.** Да се синтезира програмен фрагмент, който намира сумата от елементите над главния диагонал и произведението на елементите под главния диагонал на матрицата A[nxn], n≥1.

## Допълнителна литература


1. Dejkstra, E. W. A Discipline of Programming, Prentice-Hall, Englewood Cliffs, 1976.
2. Dejkstra, E. W. Guarded commands, nondeterminacy and the formal derivation of programs, Comm. ACM, v. 18, 1978.
3 Morgan, C.C. Programming from specifications, Prentice-Hall, 1990.
4. Грис, Д. Наука программирования, Москва, Мир, 1984.
5. Манна, З. Математически основи на информатиката, София, Наука и изкуство, 1983.
6. Тодорова, М. Синтезиране на програми, СОФТЕХ, София, 1998.

# 13

# Синтактичен анализ и намиране на стойност на израз. Търсене с връщане назад

## 13.1 Синтактичен анализ и намиране на стойност на израз

В много случаи синтаксисът на различни езикови конструкции има рекурсивна структура. За формалното описание на такива конструкции се използва широкоразпространеният език на Бекус-Наур. Например цяло число без знак може да се опише по следните два начина:

```
<цяло_без_знак> ::=  <цифра>|
                     <цяло_без_знак><цифра>
```

или

```
<цяло_без_знак> ::=  <цифра>|
                     <цифра><цяло_без_знак>
```

Да се направи синтактичен анализ на символен низ според някакви правила означава да се провери дали низът е получен според тези правила, да се определи видът на съставящите го части и връзките между тях. За целта се конструира т. нар. **дърво на синтактичния разбор**.

Например символните низове (9+7) и ((8-5)/2) са изрази според правилата:

```
<израз> ::= <цифра>|(<израз><знак><израз>)
<знак> ::= +|-|*|/;
<цифра> ::= 0|1|...|9
```

и съответните дървета на синтактичен разбор са:

```
        <израз>                                          <израз>
      /    |    \                                   /      |      \
<израз>  <знак>   <израз>                  <израз>          <знак>  <израз>
   |       |        |                     /  |   \          |        |
<цифра>    +      <цифра>            <израз> <знак> <израз>   /    <цифра>
   |                 |                  |       |       |                |
   9                 7                <цифра>   -    <цифра>           2
                                         |              |
                                         8              5
```

Задачата на синтактичния анализ не се състои в рисуване на такива дървета, а в анализ на текста и ако е правилен, да се конструира някаква структура, която определя вида на съставящите низа части и връзките между тях. В горния случай следните структури определят дърветата на синтактичния разбор:

```
expr(sign(+), expr(digit(9)), expr(digit(7)))
expr(sign(/), expr(sign(-), expr(digit(8)), expr(digit(5))),
              expr(digit(2))).
```

В тази част няма да разгледаме в пълнота задачата за синтактичния анализ на изрази. Ще дискутираме само проверката дали даден символен низ е израз в смисъла на дадени правила. Тъй като дефинициите на тези правила обикновено са рекурсивни, най-естествените решения са рекурсивните.

**Задача 111.** Да се състави рекурсивна програма, която проверява дали низ, въведен от клавиатурата, *започва* с израз, определен от правилата:

```
<израз> ::= <цифра>|(<израз><знак><израз>)
<знак> ::= +|-|*|/;
<цифра> ::= 0|1|...|9.
```

Програма Zad111.cpp решава задачата. В нея са дефинирани булевите функции:

bool formula1(); - проверява дали въведеният низ започва с формула;

bool digit(char); - проверява дали символ е цифра;

bool sign(char); - проверява дали символ е знак, според указаното правило.

```
// Program Zad111.cpp
#include <iostream.h>
bool formula1();
bool digit(char);
bool sign(char);
int main()
{if (formula1()) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
bool digit(char c)
{return c >= '0' && c <= '9';
}
bool sign(char c)
{return c == '+' || c == '-' || c == '*' || c == '/';
}
bool formula1()
{char c;
 cin >> c;
 if (c != '(') return digit(c);
 bool yes = formula1();
 if (!yes) return false;
 cin >> c;
 if (!sign(c)) return false;
 yes = formula1();
 cin >> c;
 return yes && c == ')';
}
```

**Забележка:** Ще отбележим още веднаж, че програмата отговаря положително, както за низа 8, така и за низовете 88 и 8a, както за низа (3+9), така и за низа (3+9)a-5.

Задачата за правилността на символен низ относно дадени правила ще решаваме по аналогичен начин, но чрез индекс ще следим до коя позиция е разпозната търсената форма и отговорът ще е положителен само ако низът е изчерпен.

**Задача 112.** Да се състави програма, която определя дали символен низ е израз в смисъла на следните правила:

```
<израз> ::= <израз>+<терм>|
            <израз>-<терм>|
            <терм>;
<терм> ::= <терм>*<цифра>|
           <терм>/<цифра>|
           <цифра>;
<цифра> ::= 0|1| ... |9.
```

Във функцията main се въвежда символен низ, в който се “изтриват” интервалите. Четенето на отделните символи от низа се осъществява чрез символната функция getchar:

```
char getchar()
{i++;
 if (i == len) return ' ';
 else return s[i];
}
```

която използва следните глобални променливи:

```
int i,            // индекс на текущия символ
    len;          // дължина на символния низ s
char s[100];      // символния низ, анализиран за израз
```

и връща сочения от индекса i символ, ако 0≤i<len и интервал, в противен случай.

Ще дефинираме следните рекурсивни функции:

bool expr() - реализира правилото

```
<израз> ::= <терм>|
            <израз>+<терм>|
            <израз>-<терм>;
```

bool term() - реализира правилото

```
<терм> ::= <цифра>|
```

```
              <терм>*<цифра>|
              <терм>/<цифра>;
```

Забелязваме, че тези дефиниции са ляво-рекурсивни, заради това че операциите +, -, * и / са лявоасоциативни. Дословното им реализаране ще доведе до зацикляне в неграничните случаи. Този проблем е известен като **проблем на лявата рекурсия**. Промяната на асоциативността на операциите, т.е.

```
<израз> ::= <терм>+<израз>|
            <терм>-<израз>|
            <терм>;
<терм> ::= <цифра>*<терм>|
           <цифра>/<терм>|
           <цифра>;
```

в случая решава проблема на лявата рекурсия. Програма Zad112.cpp дава едно решение на задачата.

```
// Program Zad112.cpp
#include <iostream.h>
#include <string.h>
char c;
int i, len;
char s[100];
char getchar()
{i++;
 if (i==len) return ' ';
 else return s[i];
}
bool expr();
bool term();
bool digit();
int main()
{cout << "Input an expression! ";
 char t[100];
 cin.getline(t, 100);
 len = strlen(t);
 i = -1;
 for (int j = 0; j <= len-1; j++)
```

```
      if (t[j] != ' ')
       {i++;
        s[i] = t[j];
       }
    len = i+1;
    i = -1;
    if (expr() && i+1 == len) cout << " yes \n";
    else cout << "no\n";
    return 0;
   }
   bool digit()
   {c = getchar();
    return c >= '0' && c <= '9';
   }
   bool term()
   {if (!digit()) return false;
    c = getchar();
    if (c != '*' && c != '/')
    {i--;
     return true;
    }
    return term();
   }
   bool expr()
   {if (!term()) return false;
    c = getchar();
    if (c != '+' && c != '-')
    {i--;
     return true;
    }
    return expr();
   }
```

**Задача 113.** Да се напише програма, която въвежда символен низ и ако той е правилен израз според правилата от предишната задача, пресмята стойността му.

В предишната задача проблема на лявата рекурсия решихме чрез промяна асоциативността на операциите. Това не оказа влияние на правилността на решението. Ако трябва обаче да се намери стойността на символен низ, представящ израз по новите правила, ще се получи грешен резултат (действията ще се извършват отдясно наляво).

В случая с лявата рекурсия ще се справим като запишем правилата по следния нерекурсивен начин:

<израз> ::= <терм>{<знак1><терм>}$_{опц}$

<терм> ::= <цифра>{<знак2><цифра>}$_{опц}$

<знак1> ::= +|-

<знак2> ::= *|/

Програма Zad113.cpp решава задачата.

```
// Program Zad113.cpp
#include <iostream.h>
#include <string.h>
char c;
int i, len;
char s[100];
char getchar()
{i++;
 if (i == len) return ' ';
 else return s[i];
}
bool expr(double&);
bool expr1(double, char, double&);
bool term(double&);
bool term1(double, char, double&);
bool digit(double& x)
{c = getchar();
 x = (int)c-48;
 return c >= '0' && c <= '9';
}
int main()
{cout << "Input an expression! " ;
 char t[100];
```

```
 cin.getline(t, 100);
 len = strlen(t);
 i = -1;
 for (int j = 0; j <= len-1; j++)
   if (t[j] != ' ')
    {i++;
     s[i] = t[j];
    }
 len = i+1;
 i = -1;
 double m;
 if (expr(m) && i+1 == len) cout << m << endl;
 else cout << "no \n";
 return 0;
}
bool term(double& res)
{if (!digit(res)) return false;
 c = getchar();
 if (c == ' '|| c == '+' || c == '-') {i--; return true;}
 if (c != '*' && c != '/') return false;
 return term1(res, c, res);
}
bool term1(double x, char ch, double& res)
{if (!digit(res)) return false;
 switch (ch)
 {case '*': res = x*res; break;
  case '/': res = x/res;
 }
 c = getchar();
 if (c == ' ' || c == '+' || c == '-'){i--; return true;}
 if (c != '*' && c != '/') return false;
 return term1(res, c, res);
}
bool expr(double& res)
{if (!term(res)) return false;
 c = getchar();
```

```
 if (c == ' ') {i--; return true;}
 if (c != '+' && c != '-') return false;
 return expr1(res, c, res);
}
bool expr1(double x, char ch, double& res)
{if (!term(res)) return false;
 switch (ch)
 {case '+': res = x + res; break;
  case '-': res = x - res;
 }
 c=getchar();
 if (c == ' '){i--; return true;}
 if (c != '+' && c != '-') return false;
 return expr1(res, c, res);
}
```

## 13. 2 Търсене с връщане назад

При редица задачи се поставят въпроси като “Колко начина за ... съществуват?” или “Има ли начин за ...? или възниква необходимостта от намиране на всички възможни решения, т.е. да се изчерпят всички възможни варианти за решаване на дадена задача. Широкоизползван общ метод за организация на такива търсения е **методът търсене с връщане назад** (backtracking). При него всяко от решенията (вариантите) се строи стъпка по стъпка. Частта от едно решение, построена до даден момент се нарича **частично решение**.

*Методът се състои в следното:*

- Конструира се едно частично решение.
- Проверява се дали частичното решение е общо (търсеното). Ако е така, решението се запомня или изважда и процесът на търсене или завършва, или продължава по същата схема, докато бъдат генерирани всички възможни решения.
- В противен случай се прави опит текущото частично решение да се продължи (според условието на задачата).

- Ако на някоя стъпка се окаже невъзможно ново разширяване, извършва се връщане назад към предишното частично решение и се прави нов опит то да се разшири (продължи) по друг, различен от предишния начин.

Търсенето с връщане назад се осъществява като се използва механизмът на рекурсията.

Ще го илюстрираме чрез няколко примера.

**Задача 114.** (Пътища в лабиринт) Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни  като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира всички пътища от квадратчето в най-горния ляв ъгъл до квадратчето в най-долния десен ъгъл. Пътищата (ако съществуват) да се маркират със '*'.

Мрежата ще представим чрез квадратна матрица M от символи. Квадратче (i, j) е запълнено със символа 1, ако е непроходимо и с 0 – ако е проходимо.

Програма Zad114.cpp решава задачата. В нея функцията init реализаира въвеждане на мрежата. При това се осигурява квадратче (0, 0) да е проходимо. Това не е ограничение, тъй като ако то е непроходимо, задачата няма смисъл. Функцията writelab извежда мрежата с маркирания със * път, ако съществува или съобщава, че път не съществува. Съществената част от програмата е функцията void labyrinth(int i, int j).  Тя осъществява търсенето на всички пътища от квадратче (i, j) до квадратче (n-1, n-1). /Предполагаме, че сме намерили частично решение – път от квадратче (0, 0) до квадратче (i, j)/. В главната функция main обръщението към нея се осъществява с фактически параметри 0 и 0. В labyrinth е реализиран следният алгоритъм:

- Ако квадратче (i, j) съвпада с (n-1, n-1) е намерен един път от квадратче (i, j) до квадратче (n-1, n-1), извежда се и или се завършва изпълнението, или се продължава търсенето на други пътища;

- Ако горното не е вярно, а квадратче (i, j) е извън мрежата или е непроходимо, labyrinth завършва изпълнението си;
- Ако квадратче (i, j) е вътре в мрежата и е проходимо, частичното решение се продължава като квадратче (i, j) се включва в пътя /маркира се със символа */ и продължава търсенето на всички възможни пътища от някое от четирите ограждащи (i,j) чрез стена квадратчета до квадратче (n-1, n-1), т.е.

```
labyrinth(i+1, j);
labyrinth(i, j+1);
labyrinth(i-1, j);
labyrinth(i, j-1);
```

Завършването на изпълнението на тези рекурсивни функции означава, че са невъзможни нови разширения, т.е. не съществуват други пътища от квадратче (i, j) до квадратче (n-1, n-1). Осъществява се връщане назад към предишното частично решение, като се отказваме квадратче (i, j) да принадлежи на пътя чрез възстановяване проходимостта му.

```
// Program Zad114.cpp
#include <iostream.h>
char m[20][20];
int n;
bool way = false;

void init()
{int i, j;
 do
 {cout << "n= ";
  cin >> n;
 } while(n < 1 || n > 20);
 do
 {cout << "labyrinth:\n";
  for (i = 0; i <= n-1; i++)
  for (j = 0; j <= n-1; j++)
    cin >> m[i][j];
```

```
 } while (m[0][0] != '0');
}
void writelab()
{int k, l;
 cout << endl;
 for (k = 0; k <= n-1; k++)
  {for (l = 0; l <= n-1; l++)
     cout << m[k][l] << " ";
   cout << endl;
  }
}
void labyrinth(int i, int j)
{if (i == n-1 && j == n-1)
 {m[i][j] = '*';
  way = true;
  writelab();
 }
 else
  if (i >= 0 && i <= n-1 && j >= 0 && j <= n-1)
   if(m[i][j] == '0')
   {m[i][j] = '*';
    labyrinth(i+1, j);
    labyrinth(i, j+1);
    labyrinth(i-1, j);
    labyrinth(i, j-1);
    m[i][j] = '0';
   }
}
int main()
{init();
 labyrinth(0, 0);
 if (!way) cout << "no \n";
 return 0;
}
```

**Задача 115.** Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира най-кратък път от квадратчето в най-горния ляв ъгъл до квадратчето в най-долния десен ъгъл. Пътят (ако съществува) да се маркира със '*'.

Програма Zad115.cpp решава задачата. Чрез алгоритъм, аналогичен на този от предишната задача, се генерират всички възможни пътища от квадратче (0, 0) до квадратче (n-1, n-1). От тях се избира един с най-малка дължина (брой квадратчета, включени в него). Освен масива M, програмата поддържа и двумерен масив P, в който пази мрежата с текущия най-кратък път. Тези структури са описани като глобални променливи. Като глобални са дефинирани и n – размерност на мрежата, bmin – дължина на текущия минимален път, b – дължина на текущо конструирания път и way – булева променлива, индицираща съществуването на път от квадратче (0, 0) до квадратче (n-1, n-1).

```
// Program Zad115.cpp
#include <iostream.h>
char p[20][20];
char m[20][20];
int n, bmin = 0, b = 0;
bool way = false;
void init()
{int i, j;
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 20);
 do
 {cout << "labyrinth: \n";
  for (i = 0; i <= n-1; i++)
```

```
  for (j = 0; j <= n-1; j++)
    cin >> m[i][j];
 } while (m[0][0] != '0');
}
void opt()
{int k, l;
 if (b < bmin || bmin == 0)
 {bmin = b;
  for (k = 0; k <= n-1; k++)
     for (l = 0; l <= n-1; l++)
        p[k][l] = m[k][l];
 }
}
void writelab()
{int k, l;
 if (!way) cout << "no way \n";
 else
 {cout << "yes \n";
  for (k = 0; k <= n-1; k++)
  {for (l = 0; l <= n-1; l++)
     cout << p[k][l] << " ";
   cout << endl;
  }
 }
}
void labyrinth(int i, int j)
{if (i == n-1 && j == n-1) {way = true; m[i][j]='*'; opt();}
 else
   if (i >= 0 && i <= n-1 && j >= 0 && j <= n-1)
   if (m[i][j] == '0')
   {m[i][j] = '*';
     b++;
     labyrinth(i+1, j);
     labyrinth(i, j+1);
     labyrinth(i-1, j);
     labyrinth(i, j-1);
```

```
      m[i][j] = '0';
      b--;
    }
}
int main()
{init();
 labyrinth(0, 0);
 writelab();
 return 0;
}
```

**Задача 116.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която намира и извежда всички ациклични пътища между два произволно зададени града в случай, че път между тях съществува.

Програма Zad116.cpp решава задачата. Ацикличен път е път, състоящ се от различни градове. Процедурата foundallway намира всички ациклични пътища от град i до град j в случай, че път съществува. Всеки път се записва в глобалния едномерен масив int x[100], а дължината му - в глобалната променлива s. Глобалната булева променлива way получава стойност true, ако съществува път между двата зададени града. Инициализирана е с false.

```
// Program Zad116.cpp
#include <iostream.h>
int arr[10][10] = {0};
int n, s = -1;
int x[100];
bool way = false;
void writeway()
{for (int i = 0; i <= s; i++)
  cout << x[i] << " ";
 cout << endl;
```

```
}
bool member(int x, int n, int* a)
{if (n == 1) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
void foundallway(int i, int j)
{if (i == j)
 {way = true;
  s++;
  x[s] = i;
  writeway();
 }
 else
 {s++;
  x[s] = i;
  for (int k = 0; k <= n-1; k++)
    if (arr[i][k] == 1 && !member(k, s+1, x))
    {arr[i][k] = 0; arr[k][i] = 0;
     foundallway(k, j);
     arr[i][k] = 1; arr[k][i] = 1;
     s--;
    }
   }
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 for (i = 0; i <= n-2; i++)
   for (int j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
```

```
 int j;
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
 } while (i < 0 || i > 9 || j < 0 || j > 9);
 foundallway(i, j);
 if (!way) cout << "no \n";
 return 0;
}
```

**Задача 117.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която намира и извежда ацикличен път с минимална дължина между два произволно зададени града в случай, че път съществува.

Използван е подход аналогичен на този в Задача 115.

```
// Program Zad117.cpp
#include <iostream.h>
int arr[10][10] = {0};
int n, s = -1, smin = 0;
int x[100], xmin[100];
bool way = false;
void writeway()
{if (!way) cout << "no way \n";
 else
 {cout << "yes \ n";
  for (int i = 0; i <= smin - 1; i++)
   cout << xmin[i] << " ";
  cout << endl;
 }
}
void opt()
{if (s+1 < smin || smin == 0)
```

```
 {smin = s + 1;
  for (int i = 0; i <= smin-1; i++)
    xmin[i] = x[i];
 }
}
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
void foundminway(int i, int j)
{if (i == j)
 {way = true;
  s++; x[s] = i;
  opt();
 }
 else
 {s++; x[s] = i;
  for (int k = 0; k <= n-1; k++)
    if (arr[i][k] == 1 && !member(k, s+1, x))
    {arr[i][k] = 0; arr[k][i] = 0;
     foundminway(k,j);
     s--;
     arr[i][k] = 1; arr[k][i] = 1;
    }
 }
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 for (i = 0; i <= n-2; i++)
   for (int j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
    cin >> arr[i][j];
```

```
      arr[j][i] = arr[i][j];
     }
    int j;
    do
    {cout << "start and final towns: ";
     cin >> i >> j;
    } while (i < 0 || i > 9 || j < 0 || j > 9);
     foundminway(i, j);
     writeway();
     return 0;
   }
```

## Задачи

**Задача 1.** Да се направи програма, която анализира символен низ въведен от клавиатурата и определя дали той е израз в смисъла на следната граматика:

```
<израз> ::= <израз>+<терм>|
            <израз>-<терм>|
            <терм>;
<терм> ::= <терм>*<буква>|
           <терм>/<буква>|
           <буква>;
<буква> ::= a|b|c ... |z.
```

**Задача 2.** Да се напише програма, която проверява дали символен низ, въведен от клавиатурата е израз съгласно формулата:

```
<израз> ::= <цифра>|
            f(<израз>)|s(<израз>)|p(<израз>)
<цифра> ::= 0|1|...|9
```

и ако това е така, пресмята стойността на израза, ако f(x), s(x) и p(x) намират съответно x!, x+1 и x-1.

**Задача 3.** Да се състави програма, която проверява дали низ е израз, определен от формулите:

```
<израз> ::= <цифра>|(<израз><знак><израз>)
<знак> ::= +|-|*|/;
```

<цифра> ::= 0|1|...|9

и ако това е така, намира стойността му.

**Задача 4.** Да се напише програма, която проверява дали символен низ, въведен от клавиатурата е израз съгласно формулата:

<израз> ::= <цифра>|
pow(<израз>,<израз>)|gcd(<израз>,<израз>)

<цифра> ::= 0|1|...|9

и ако това е така, пресмята стойността на израза, ако pow(x, y) и gcd(x, y) намират съответно x на степен y и най-големия общ делител на x и y (Ако някое от числата x или y е 0, за gcd(x,y) да се приеме другото или 0, ако и x, и y са 0).

**Задача 5.** Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира най-краткия път от квадратче (0, 0) до квадратче (n-1, n-1), който минава през произволно зададено проходимо квадратче на мрежата. Пътят да се маркира със '*' (ако съществува).

**Задача 6.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която намира и извежда всички ациклични пътища с указана дължина между два произволно зададени града в случай.

**Задача 7.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Ацикличен път, в който началния и крайния град съвпадат, се нарича цикъл. Да се напише програма, която намира и извежда всички цикли за произволно зададен град в случай, че цикъл съществува.

**Задача 8.** Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че $a_{ij}$ е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Ацикличен път, в който минава през всички върхове, ще наричаме Хамилтонов цикъл. Да се напише програма, която намира и извежда всички Хамилтонови цикли за зададените градове.

## Допълнителна литература

1. Рейнгольд З., Нивергальт Н. Део, Комбинаторные алгоритмы. Теория и практика, Москва, Мир, 1980.

2. Узерелл Ч., Этюды для программистов, Москва, Мир, 1982.

3. Тодорова М., Езици за функционално и логическо програмиране. Логическо програмиране, София, СОФТЕХ, 1998.

# 14

# Класове

Класовете са нови типове данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ тип или да представят напълно нов тип данни.

Класовете са подобни на структурите, даже може да се каже, че в някои отношения са почти идентични. В C++ класът може да се разглежда като структура, на която са наложени някои ограничения по отношение на правата на достъп. Отначало ще разгледаме основното, което е приложимо както за класовете, така и за структурите. Затова ще останем в познатите означения на структурите.

## 14.1 Пример за програма, която дефинира и използва клас

Основен принцип на процедурното програмиране е модулния. Програмата се разделя на "подходящи" взаимносвързани части (функции, модули), всяка от които се реализира чрез определени средства. Важен е обаче начинът, по който да става определянето на частите и връзките помежду им. Целта е, следващи промени в представянето на данните да не променят голям брой от модулите на програмата. Разсъждения в тази посока довеждат до подхода *абстракция със структури от данни*, който вече разгледахме. Ще напомним, че при него методите за използване на данните се разделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с "абстрактни данни" – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени **конструктори, мутатори** и **функции за достъп**, които реализират "абстрактните данни" по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракцията:

Добра реализация на подхода е тази, при която всяко ниво използва единствено средствата на предходното. Предимствата са, че възникнали промени на едно ниво ще се отразят само на следващото над него. Например, промяна на представянето на данните ще доведе до промени единствено на реализацията на някои от конструкторите, мутаторите или функциите за достъп.

Да се върнем към задачата за рационално-цифрова аритметика. Като използваме подхода абстракция със структури от данни искаме да дефинираме тип данни “рационално число”, след което да го използваме за събиране, изваждане, умножение и деление на рационални числа.

След анализ на правилата, реализиращи тези операции, в глава 11 стигнахме до необходимостта от реализирането на следните примитивни операции за работа с рационални числа:

- конструиране на рационално число по зададени две цели числа, представящи съответно неговите числител и знаменател;
- извличане на числителя на дадено рационално число;
- извличане на знаменателя на дадено рационално число.

Към тях ще добавим и функциите:

- промяна на стойността на рационално число чрез въвеждане, например;
- извеждане на рационално число.

Реализирането на подхода абстракция със структури от данни в този случай показва следните четири нива на абстракция

Ще започнем с реализирането на нивата отдолу нагоре.

*Избор на представяне на рационално число*

Тъй като рационалното число е частно на две цели числа, можем да го определим чрез структурата:

```
struct rat
{int numer;
 int denom;
};
```

където полето numer означава числителя, а полето denom – знаменателя на рационално число. Тези две полета се наричат **член-данни**, само **данни** или още **абстрактни данни** на структурата. Те определят множеството от стойности на типа rat, който дефинираме.

Трябва да добавим и някакви операции и вградени функции, които да могат да се изпълняват над данни от тип rat. Това ще постигнем с реализацията на следващите две нива на абстракция, определени по-горе.

*Реализиране на примитивните операции*

Като компоненти на структурата rat ще добавим набор от примитивни операции: конструктори, мутатори и функции за достъп. Ще ги реализираме като член-функции.

*а) конструктори*

Конструкторите са член-функции, чрез които се инициализират променливите на структурата. Имената им съвпадат с името на структурата. Ще дефинираме два конструктора на структурата rat:

rat() – конструктор без параметри и

rat(int, int) – конструктор с два цели параметъра.

Първият конструктор се нарича още **конструктор по подразбиране**. Използва се за инициализиране на променлива от тип rat, когато при дефиницията й не са зададени параметри. Ще го дефинираме така:

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

**Пример:** След дефиницията

```
rat p = rat();
```

или съкратено

```
rat p;
```

p се инициализира с рационалното число 0/1.

Вторият конструктор

```
rat::rat(int x, int y)
{numer = x;
 denom = y;
}
```

позволява променлива величина от тип rat да се инициализира с указана от потребителя стойност.

**Примери:** След дефиницията

```
rat p = rat(1,3);
```

p се инициализира с 1/3, а дефиницията

```
rat q(2,5);
```

инициализира q с 2/5.

Ще отбележим, че и двата конструктора имат едно и също име, но се различават по броя на параметрите си. В този случай се казва, че функцията rat е **предефинирана**.

Декларацията на структура може да съдържа, но може и да не съдържа конструктори.

*б) мутатори*

Това са функции, които променят данните на структурата. Ще дефинираме мутатора read(), който въвежда от клавиатурата две цели числа (второто различно от нула) и ги свързва с абстрактните данни numer и denom.

```
void rat::read()
{cout << “numer: “;
 cin >> numer;
 do
 {cout << “denom: “;
  cin >> denom;
 } while (denom == 0);
}
```

След обръщението

```
p.read();
```

стойността на p *се променя* като полетата й numer и denom се свързват с въведените от потребителя стойности за числител и знаменател съответно.

*в) функции за достъп*

Тези функции **не променят** член-данните на структурата, а само извличат информация за тях. Последното е указано чрез използването на запазената дума const, записана след затварящата скоба на формалните параметри и пред знака ;. Ще дефинираме следните функции за достъп:

```
int get_numer() const;
int get_denom() const;
void print() const;
```

Първата от тях извлича числителя, втората – знаменателя, а третата извежда върху екрана рационалното число numer/denom. Реализациите им имат вида:

```
int rat::get_numer() const
{return numer;
}
int rat::get_denom() const
{return denom;
}
void rat::print() const
{cout << numer << “/” << denom << endl;
}
```

След включване на *прототипите* на тези конструктори, мутатори и функции за достъп във фигурните скоби на декларацията на структурата rat, получаваме:

```
struct rat
{int numer;
 int denom;
 // конструктори
 rat();
 rat(int, int);
 // мутатор
 void read();
 // функции за достъп
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

След направените дефиниции са възможни следните действия над рационални числа:

```
// p се инициализира с 0/1, q – с 1/6, а r – с 5/9
   rat p, q(1,6), r=rat(5,9);
// p се извежда чрез данновите полета на структурата rat
   cout << p.numer << "/"
        << p.denom << endl;
// q се извежда като се използват
// функциите за достъп до компонентите му
   cout << q.get_numer() << "/"
```

```
        << q.get_denom() << endl;
// q се извежда чрез функцията за достъп print()
   q.print();
// p се модифицира чрез мутатора read()
   p.read();
```

С това завършихме реализирането на двете най-долни нива на абстракция и преминаваме към следващото ниво.

*Реализиране на правилата за рационално-цифрова аритметика*

Като използваме дефинираните конструктори, мутатори и функции за достъп, ще реализираме функциите:

```
rat sum(rat const &, rat const &);
rat sub(rat const &, rat const &);
rat prod(rat const &, rat const &);
rat quot(rat const &, rat const &);
```

извършващи рационално-числовата аритметика. Функцията sum може да се дефинира по следния начин:

```
rat sum(rat const& r1, rat const& r2)
{rat r(r1.get_numer()*r2.get_denom() +
       r2.get_numer()*r1.get_denom(),
       r1.get_denom()*r2.get_denom());
 return r;
}
```

Другите функции се реализират по аналогичен начин.

Ще отбележим, че по подразбиране, членовете на структурата (член-данни и член-функции) са видими навсякъде в областта на структурата. Това позволява член-данните да бъдат използвани както от примитивните конструктори, мутатори и функции за достъп така и от функциите, реализиращи рационално-числова аритметика.

Например, функцията sum, дефинирана по-горе може да се реализира и така:

```
rat sum(rat const& r1, rat const& r2)
{rat r(r1.numer*r2.denom + r2.numer*r1.denom,
       r1.denom*r2.denom);
 return r;
}
```

Нещо повече, освен чрез мутаторите, член-данните могат да бъдат модифицирани и от външни функции.

Последното противоречи на идеите на подхода абстракция със структури от данни, в основата на който лежи независимостта на използването от представянето на структурата от данни. Това води до идеята да се забрани на модулите от трето и четвърто ниво пряко да използват средствата от първо ниво на абстракция.

Езикът C++ позволява да се ограничи свободата на достъп до членовете на структурата като се поставят подходящи спецификатори на достъп в декларацията й. Такива спецификатори са private и public. Записват се като етикети. Всички членове, следващи спецификатора на достъп private, са достъпни само за член-функциите в декларацията на структурата. Всички членове, следващи спецификатора на достъп public, са достъпни за всяка функция, която е в областта на структурата. Ако са пропуснати спецификаторите на достъп, всички членове са public (както е в случая). Има още един спецификатор на достъп – protected, който е еднакъв със спецификатора private, освен ако структурата не е част от йерархия на класовете, което ще разгледаме по-късно.

С цел реализиране на идеите на подхода абстракция със структури от данни, ще променим дефинираната по-горе структура по следния начин:

```
struct rat
{private:
 int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

По такъв начин позволяваме член-данните numer и denom да се използват единствено от член-функциите на структурата rat. Операторът

```
cout << p.numer << "/"
```

```
        << p.denom << endl;
```

вече е недопустим.

Следва програмата, която решава задачата.

```
#include <iostream.h>
struct rat
{private:
 int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
rat::rat()
{numer = 0;
 denom = 1;
}
rat::rat(int x, int y)
{numer = x;
 denom = y;
}
void rat::read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 }while(denom==0);
}
int rat::get_numer() const
{return numer;
}
int rat::get_denom() const
{return denom;
}
```

```
void rat::print() const
{cout << numer << "/" << denom << endl;
}
rat sum(rat const &, rat const &);
rat sub(rat const &, rat const &);
rat prod(rat const &, rat const &);
rat quot(rat const &, rat const &);
int main()
{rat p(1,4), q(1,2);
 p.print();
 q.print();
 cout << "sum:\n";
 sum(p,q).print();
 cout << "subtraction:\n";
 sub(p,q).print();
 cout << "product:\n";
 prod(p,q).print();
 cout << "quotient:\n";
 quot(p,q).print();
 return 0;
}
rat sum(rat const& r1, rat const& r2)
{rat r(r1.get_numer()*r2.get_denom()+
       r2.get_numer()*r1.get_denom(),
       r1.get_denom()*r2.get_denom());
 return r;
}
rat sub(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_denom()-
       r2.get_numer()*r1.get_denom(),
       r1.get_denom()*r2.get_denom());
 return r;
}
rat prod(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_numer(),
       r1.get_denom()*r2.get_denom());
 return r;
}
```

```
rat quot(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_denom(),
       r1.get_denom()*r2.get_numer());
 return r;
}
```

Като се използват функциите за рационално-числова аритметика, могат да се реализират различни приложения. Забелязваме обаче, че тази реализация не съкращава рационални числа. За преодоляването на този недостатък е достатъчно да променим конструктора с параметри. За целта реализираме разделяне на числителя x и знаменателя y на най-големия общ делител на abs(x) и abs(y). Новият конструктор има вида:

```
rat::rat(int x, int y)
{if (x == 0 || y==0)
 {numer = 0;
 denom = 1;
 }
 else
 {int g = gcd(abs(x), abs(y));
  if (x>0 && y>0 || x<0 && y<0)
  {numer = abs(x)/g;
   denom = abs(y)/g;
  }
  else
  {numer = -abs(x)/g;
   denom = abs(y)/g;}
  }
}
```

където int gcd(int x, int y) е известната вече функция за намиране на най-големия общ делител на две естествени числа.

Ще отбележим, че ако в горната програма заменим запазената дума struct с class, програмата няма да промени поведението си. Така дефинирахме класа rat:

```
class rat
{private:
    int numer;
    int denom;
 public:
```

```
        rat();
        rat(int, int);
        void read();
        int get_numer() const;
        int get_denom() const;
        void print() const;
    };
```

а дефиницията

```
    rat p, q=rat(1,7), r(-2,9);
```

определя три негови **обекта**: p, инициализиран с рационалното число 0/1; q, инициализиран с 1/7 и r, инициализиран с -2/9.

Спецификаторът private, забранява използването на член-данните numer и denom извън класа. Получава се *скриване* на информация, което се нарича още **капсолиране на информация**. Член-функциите на класа rat са обявени като public. Те са видими извън класа и могат да се използват от външни функции. Затова public-частта се нарича още **интерфейсна част на класа** или само **интерфейс**. Чрез нея класът комуникира с външната среда. Освен функции, интерфейсът може да съдържа и член-данни, но засега ще се стараем това да не се случва.

*Ще отбележим, че конструкторите се използват само когато се създават обекти. Опитите за* ***промяна*** *на обект чрез обръщение към конструктор предизвикват грешки.*

**Пример:**

```
    rat q(1,7);     // коректно
    q.rat();        // предизвиква грешка
    q(2,9);         // предизвиква грешка
    q.rat(3,4);     // предизвиква грешка.
```

Забележете, езикът C++ дефинира структурите и класовете почти идентично. Съществената разлика е свързана със спецификаторите на достъп. По подразбиране членовете на структура имат public (публичен) достъп, а членовете на клас – private (частен) достъп. Все пак възниква въпросът: *Защо да има две различни конструкции struct и class, когато разликите са толкова малки?* Причината е свързана с мобилността на програмите, с цел да се запази съвместимостта между езиците C и C++. Бярне Страуструп обяснява, че причината е по-скоро културна, отколкото техническа. Той препоръчва структурите да се използват само когато се реализират

свойства, които са прости и включват малки “натоварвания”. По-точно, когато реализираната структура от данни е идентична с интерфейса си. В останалите случаи да се използват класове.

## 14.2 Дефиниране на класове

Класовете осигуряват механизми за създаване на напълно нови типове данни, които могат да бъдат интегрирани в езика, а също за обогатяване възможностите на вече съществуващи типове. Дефинирането на един клас се състои от две части:

- декларация на класа и
- дефиниция на неговите член-функции (методи).

### 14.2.1. Декларация на клас

Декларацията на клас се състои от заглавие и тяло. Заглавието започва със запазената дума class, следвано от името на класа. Тялото е заградено във фигурни скоби. След скобите стои знакът “;” или списък от обекти. В тялото на класа са декларирани членовете на класа (член-данни и член-функции) със съответните им нива на достъп. Фиг. 14.1 илюстрира *непълно* (но достатъчно за целите на настящите разглеждания) синтаксиса на декларацията на клас.

**Декларация на клас**

```
<декларация_на_клас> ::= <заглавие> <тяло>
<заглавие> ::= class [<име_на_клас>]опц
<тяло> ::= {<декларация_на_член>;
            {<декларация_на_член>;}опц
           }[<списък_от_обекти>]опц;
<декларация_на_член> ::=
     <декларация_на_конструктор>|<декларация_на_мутатор>|
     <декларация_на_функция_за_достъп>|<декларация_на_данна>
<декларация_на_конструктор> ::=
     [<спецификатор_на_достъп>:]опц<име_на_клас>(<параметри>)
<декларация_на_мутатор> ::=
     [<спецификатор_на_достъп>:]опц <тип>
                <име_на_мутатор>(<параметри>)
<декларация_на_функция_за_достъп> ::=
```

```
        [<спецификатор_на_достъп>:]опц <тип>
                    <име_на_функция_за_достъп>(<параметри>) const;
   <спецификатор_на_достъп> ::= private | public | protected
   <параметри> :: <празно> | void |
                    <параметър> {, <параметър>}опц
   <параметър> ::= <тип> [ &|опц * [const]опц] опц
   <декларация_на_данна>      ::=      <тип>      <име_на_данна>{,
<име_на_данна>}опц
   <тип> ::= <име_на_тип>|<дефиниция_на_тип>
   <списък_от_обекти> ::=
       <обект> [= <име_на_клас>(<фактически_параметри>)]опц
               {,<обект>[=<име_на_клас>(<фактически_параметри>)]опц
}опц
               {, <обект>(<фактически_параметри>)}опц
               {, <обект> = <вече_дефиниран_обект>}опц
```

където <име_на_клас>, <име_на_мутатор>, <име_на_данна>, <обект> и <име_на_функция_за_достъп> са идентификатори, а <фактически_параметри> е определено в Глава 8.

Фиг. 14.1 Декларация на клас

За имената на класовете важат същите правила, които се прилагат за имената на всички останали типове и променливи. Също като при структурите името на класа може да бъде пропуснато. Ще напомним, че използването на долния индекс “опц” означава, че означението е от езика на Бекус-Наур за описание на синтаксиса на език за програмиране.

Имената на членовете на класа са локални за него, т.е. в различни класове в рамките на една програма могат да се дефинират членове с еднакви имена. Член-данни от един и същ тип могат да се изредят, разделени със запетая и предшествани от типа им.

**Пример:**

```
class point
{private:
  double x, y;    // x и y са член-данни на класа prat
 public:
  point(double, double);
```

```
    void read();       // мутатор със същото име като на класа rat
    int get_x() const;
    int get_y() const;
    void print() const;   // със същото име като на класа rat
  }p=point(2,7), q(-2,3), r=q;
```

Препоръчва се член-данните да се декларират в нарастващ ред по броя на байтовете, необходим за представянето им в паметта. Така за повечето реализации се получава оптимално изравняване до дума.

**Забележка:** Типът на член-данна на клас **не може** да съвпада с името на класа, но типът на член-функция на клас **може** да съвпада с името на класа.

В тялото, някои декларации на членове могат да бъдат предшествани от **спецификаторите на достъп** private, public или protected. Областта на един спецификатор на достъп започва от спецификатора и продължава до следващия спецификатор. Подразбиращ се спецификатор за достъп е private. Един и същ спецификатор на достъп може да се използва повече от веднъж в декларация на клас.

Препоръчва се, ако секция public съществува, да бъде първа в декларацията, а секцията private да бъде последна в тялото на класа.

*Достъпът до членовете на класовете може да се разгледа на следните две нива:*

- По отношение на *член-функциите в класа* е в сила, че те имат достъп до всички членове на класа. При това не е необходимо тези компоненти да се предават като параметри. Този режим на достъп се нарича **режим на пряк достъп**. Поради тази причина функциите rat(), read(), print(), get_numer() и get_denom() са без параметри. Освен това член-функцията print() може да бъде дефинирана и по следния начин:

```
  void rat::print() const
  {cout << get_numer() << "/" <<
        << get_denom() << endl;
  }
```

или

```
  void rat::print() const
```

```
{cout << this->get_numer() << "/" <<
      << this->get_denom() << endl;
}
```

Смисълът на this ще бъде обяснен в т. 14.4.5, а на -> - в т.14.5.

- По отношение на *функциите, които са външни за класа*, режимът на достъп са определя от начина на декларираме на членовете.

Членовете на даден клас, декларирани като private (декларирани след запазената дума private) са видими (достъпни) само в рамките на класа. Външните функции нямат достъп до тях. По подразбиране членовете на класовете са private. Това позволява декларацията на класа rat да запишем и по следния начин:

```
class rat
{int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

Чрез използването на членове, обявени като private, се постига скриване на членове за външната за класа среда. Процесът на скриване се нарича още **капсолиране на информацията**.

Членовете на клас, които трябва да бъдат видими извън класа (да бъдат достъпни за функции, които не са методи на дадения клас) трябва да бъдат декларирани като public (декларирани след запазената дума public). Всички методи на класа rat са декларирани като public и следователно могат да се използват навсякъде в програмата за работа с рационални числа.

Освен като private и public, членовете на класовете могат да бъдат декларирани и като protected. Тъй като този спецификатор на достъп има отношение към производните класове и процеса на наследяване, разглеждането му засега ще бъде отложено. Ще отбележим, че ако в класа rat заменим private с protected, поведението на класа няма да се промени.

### 14.2.2 Дефиниране на методите на клас

След декларирането на клас, трябва да се дефинират неговите методи. Дефинициите са аналогични на дефинициите на функции, но името на метода се предшества от името на класа, на който принадлежи метода, следвано от оператора за принадлежност :: (Нарича се още оператор за област на действие). Такива имена се наричат **пълни**. (Операторът :: е ляво-асоциативен и с един и същ приоритет със (), [] и ->). На Фиг. 14.2 е даден синтаксисът на дефиницията на метод на клас.

**Дефиниция на метод на клас**

<дефиниция_на_метод_на_клас> ::=

[<тип>]$_{опц}$ <име_на_клас>::<име_на_функция>(<параметри>) [const]$_{опц}$

{<тяло>}

<тяло> ::= <редица_от_оператори_и_дефиниции>

където <име_на_клас> и <име_на_функция> са идентификатори, а <параметри> се определя както в дефиниция на функция.

Фиг. 14.2 Дефиниция на метод на клас

Ще отбележим, че дефиницията на конструктор **не започва** с <тип>, а запазената дума const може да присъства само в дефинициите на функциите за достъп. Добрият стил на програмиране изисква използването на const в дефинициите на функциите за достъп и също в техните декларации. Ако се пренебрегне това изискване, могат да се създадат класове, които да не могат да се използват от други програмисти.

**Пример:** Нека искаме да използваме класа rat, но програмистът му е забравил или нарочно не е декларирал член-функцията print() като const и rat има вида:

```
class rat
{private:
  ...
 public:
  ...
  void print();
};
```

Нека декларираме класа prat, използващ класа rat, коректно, т.е. функциите за достъп обявяваме като const.

```
class prat
{private:
 int a;
 rat p;  // използване на класа rat
 ...
 public:
 ...
 void print() const;
};
```

където

```
void prat::print() const
{cout << a << endl;
 p.print(); // тази print() е член-функцията на класа rat
};
```

Компилаторът ще съобщи за грешка в обръщението p.print(), защото p е обект на класа rat, а член-функцията rat::print() не е декларирана като const. Компилаторът предполага, че p.print() може да модифицира p. Но p е член-данна на prat, а prat::print() е const, с което твърдо е обещава да не го модифицира.

Обикновено дефинициите на методите са разположени веднага след декларирането на класа, на който те са членове. Възможно е обаче, дефинициите на методите на един клас да бъдат част от декларациите на този клас, т.е. в декларациите на член-функциите на класа могат да се зададат не само прототипите им, но и техните тела.

**Пример:** Класът rat може да бъде дефиниран и по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat()
  {numer = 0;
   denom = 1;
  }
  rat(int a, int b)
  {if (a == 0 || b==0)
```

```
      {numer = 0;
       denom = 1;
      }
      else
      {int g = gcd(abs(a), abs(b));
       if (a>0 && b>0 || a<0 && b<0)
       {numer = abs(a)/g;
        denom = abs(b)/g;}
       else
       {numer = - abs(a)/g;
        denom = abs(b)/g;
       }
      }
     }
    void read()
    {cout << "numer: ";
     cin >> numer;
     do
     {cout << "denom: ";
      cin >> denom;
     } while (denom == 0);
    }
    int get_numer() const
    {return numer;
    }
    int get_denom() const
    {return denom;
    }
    void print() const
    {cout << numer << "/" << denom << endl;
    }
   };
```

В този случай обаче член-функциите се третират като **вградени (inline) функции**.

**Допълнение (вградени функции)** С цел повишаване на бързодействието, езикът C++ поддържа т.нар. вградени функции. Кодът на тези функции не се съхранява на едно място, а се копира

на всяко място в паметта, където има обръщение към тях. Използват се като останалите функции, но при декларирането и дефинирането им заглавието им се предшества от модификатора inline.

**Пример:**

```
#include <iostream.h>
inline int f(int, int); // декларация на вградената функция f
void main()
{cout << f(1,5) << endl;
}
inline int f(int a, int b) // дефиниция на вградената функция f
{return (a+b)*(a-b);
}
```

Ще добавим, че дефиницията на вградена функция трябва да се намира в същия файл, където се използва, т.е. не е възможна разделна компилация, тъй като компилаторът няма да разполага с кода за вграждане. Използването на вградени функции води до икономия на време, за сметка на паметта. Затова се препоръчва използването им само при “кратки” функции. Ще отбележим също, че модификаторът inline е само заявка към компилатора, която може да бъде, но може и да не бъде изпълнена. Възможно е компилаторът да откаже вграждане, ако реши, че функцията е прекалено голяма или има други причини, възпрепятстващи вграждането. Ограниченията за вграждане зависят от конкретния компилатор.

Често член-функциите се реализират като вградени функции. Това увеличава ефективността на програмата, използваща класа. Декларацията на вградени член-функции може да се осъществи и по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat();
  rat(int, int);
  void read();
  int get_numer() const;
  int get_denom() const;
```

```
    void print() const;
  };
  inline rat::rat()
  {numer = 0;
   denom = 1;
  }
  inline rat::rat(int x, int y)
  ...
  inline void rat::read()
  ...
  inline int rat::get_numer() const
  ...
  inline int rat::get_denom() const
  ...
  inline void rat::print() const
  ...
```

Телата на някои от член-функциите са пропуснати, тъй като вече са известни.

Ще отбележим също, че в тялото на дефиницията на член-функция явно не се указва обектът, върху който тя ще се приложи. Този обект участва неявно - чрез член-данните на класа. Заради това се нарича **неявен параметър,** а член-данните – **абстрактни данни**. Връзката между неявния параметър и обект ще бъде показана в т. 3. Параметри, които участват явно в дефиницията на член-функция се наричат **явни**. *Всяка член-функция има точно един неявен параметър и нула или повече явни.*

### 14.2.3 Област на класовете

За разлика от функциите, класовете могат да се декларират на различни нива в програмата: *глобално* (ниво функция) и *локално* (вътре във функция или в тялото на клас).

Областта на глобално декларирен клас започва от декларацията и продължава до края на програмата. Примерите досега бяха с такива класове.

Ако клас е декларирен във функция, всички негови член-функции трябва да са вградени (inline). В противен случай ще се получат функции, дефинирани във функция, което не е възможно.

**Пример:**

```
void f(int i, int* p)
{int k;
 class CL
 {public:
   // всички методи са дефинирани в тялото на класа
  ...
  private:
  ...
 };
// тяло на функцията f
 CL x;
 ...
}
```

Областта на клас, дефиниран във функция, е функцията. Обектите на такъв клас са видими само в тялото на функцията.

Възможно е използването на обекти (в широкия смисъл на думата) с еднакви имена. В сила е правилото, че в областта си локалният обект скрива нелокалния.

Не е възможно в тялото на локално дефиниран клас да се използва функцията, в която класът е дефиниран.

**Пример:** Нека сме в означенията на горния пример.

```
void f(...)
{...
 class cl
 {
   // не може да се използва функцията f
 };
 ...
}
```

## 14.3 Обекти

След като даден клас е дефиниран, могат да бъдат създавани негови екземпляри, които се наричат **обекти**. Връзката между клас и обект в езика C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се

състоят от множество **компоненти** (член-данни и член-функции). На Фиг. 14.3 е даден синтаксисът на дефиниция на обект на клас.

**Дефиниция на обект на клас**

<дефиниция-на_обект_на_клас> ::=

<име_на_клас> <обект> [=<име_на_клас>(<фактически_параметри>)]$_{опц}$

{, <обект>[=<име_на_клас>(<фактически_параметри>)]$_{опц}$ }$_{опц}$

{, <обект>(<фактически_параметри>)}$_{опц}$

{, <обект> = <вече_дефиниран_обект>}$_{опц}$;

<обект> ::= <идентификатор>

където <фактически_параметри> е определено в Глава 8.

Фиг. 14.3 Дефиниция на обект на клас

*Когато за даден клас явно са дефинирани конструктори,* при всяко дефиниране на обект на класа те автоматично се извикват с цел да се инициализира обектът. Ако дефиницията е без явна инициализация (например rat p;), дефинираният обект се инициализира според дефиницията на конструктора по подразбиране, ако такъв е определен, и се съобщава за грешка в противен случай. Ако дефиницията е с явна инициализация, обръщението към конструкторите трябва да бъде коректно.

**Пример:** Дефиницията

rat p, q(2,3), r=rat(3,8);

определя три обекта: p, инициализиран с рационалното число 0/1, q, инициализиран с 2/3 и r, инициализиран с 3/8. Тя е добре оределена, тъй като класът rat има конструктор по подразбиране и двуаргументен конструктор. Ако елиминираме конструктора по подразбиране в класа rat, горната дефиниция ще съобщи за грешка заради p. Валидна е обаче дефиницията:

rat q(2,3), r=rat(3,8);

Ако се откажем и от другия конструктор, последната дефиниция също ще стане невалидна.

*Когато за даден клас явно не е дефиниран конструктор*, реализацията автоматично генерира **подразбиращ се конструктор**. Този конструктор изпълнява редица действия, като заделяне на памет за

обектите, инициализиране на някой системни променливи и др. Дефиницията на обект от този клас трябва да е без явна инициализация.

**Пример:**

```
#include <iostream.h>
class pom
{private:
  int a;
 public:
  int b;
  void read();
  void print() const;
};
void main()
{pom x;  // инициализация според подразбиращия се
         //конструктор, генериран от компилатора на C++
 x.read();
 x.print();
}
void pom::print()const
{cout << "a= "<< a << " b=" << b << endl;
}
void pom::read()
{cout << "a= ";
 cin >> a;
 cout << "b= ";
 cin >> b;
}
```

В случая обектът x се инициализира с неопределена стойност. Опитите за инициализацията му като структура предизвикват грешки.

Декларацията на клас не заделя памет за него. Памет се заделя едва при дефинирането на обект от класа. Дефиницията

```
rat p, q(2, 3), r = rat(3, 8);
```

заделя за обектите p, q и r по 8 байта ОП (по 4В за всяка от данните им numer и denom).

Достъпът до компонентите на обектите (ако е възможен) се осъществява чрез задаване на името на обекта и името на данната

или метода, разделени с точка (Фиг. 14.4). Изключение от това правило правят конструкторите (Фиг. 14.5).

**Достъп до компонента на обект**

<компонента_на_обект> ::= <обект>.<данна>|
<обект>.<име_на_член_функция>()|
<обект>.<име_на_член_функция>(<параметри>)

<име_на_член_функция> е <идентификатор>, означаващ име на мутатор или име на функция за достъп.

Фиг. 14.4 Достъп до компонента на обект

**Пример:**

rat p(1,2), q;

p.get_numer() // достъп до член-функцията get_numer() за обекта p

q.get_numer() // достъп до член-функцията get_numer() за обекта q

Ще отбележим също, че на практика обектите p и q нямат свои копия на метода get_numer(). И двете обръщения се отнасят за един и същ метод, но при първото обръщение се работи с данните за обекта p, а при второто – с данните за обекта q.

При създаването на обекти на един клас кодът на методите на този клас не се копира във всеки обект, а се намира само на едно място в паметта.

Естествено възниква въпросът *по какъв начин методите на един клас "разбират" за кой обект на този клас са били извикани*. Отговорът на този въпрос дава указателят this. Всяка член-функция на клас поддържа допълнителен формален параметър - указател с име this и от тип <име_на_клас>*. За да разберем точно как става това, ще разгледаме как компилаторът на C++ обработва член-функция и обръщение към член-функция на клас. Извършват се следните преобразувания:

а) Всяка член-функция на даден клас се транслира в обикновена функция с уникално име и един допълнителен параметър – указателят this.

**Пример:** Функцията

```
void rat::print()
{cout << numer << "/" << denom << endl;
}
```

се транслира в

```
void print_rat(rat* this)
{cout << this->numer << "/" << this->denom << endl;
}
```

б) Всяко обръщение към член-функция се транслира в съответствие с преобразуването от а).

**Пример:** Обръщението

```
p.print();
```

се транслира в

```
print_rat(&p);
```

Указателят this може да се използва явно в кода на съответната член-функция, макар че е глупаво да се напише:

```
void rat::print()
{cout << this->numer << "/" << this->denom << endl;
}
```

Като приложение на указателя this ще реализираме функцията

```
rat sum(rat const &, rat const &);
```

като член-функция на класа rat. За целта ще включим псевдонима й

```
rat sum(rat const &, rat const &);
```

в public-секцията на тялото на rat и ще я дефинираме по следния начин:

```
rat rat::sum(rat const & r1, rat const & r2)
{numer = r1.numer*r2.denom+r2.numer*r1.denom;
 denom = r1.denom*r2.denom;
 return *this;
}
```

Нека

```
rat p=rat(1,4), r(1,2), q=rat(1,4);
```

Фрагментът

```
r.sum(p.sum(p, r), q);
r.print();
p.print();
```

съобщава 6/8 за стойност на p и 32/32 - за стойност на r. Обръщението p.sum(p, r) намира сумата на рационалните числа p и r

и я свързва с обекта p, а r.sum(p.sum(p, r), q) събира полученото рационално число с q и свързва резултата с обекта r.

*Обекти от един и същ клас могат да се присвояват един на друг. Присвояването може да е и на ниво инициализация (фиг. 14.3).*

**Пример:** Допустими са дефинициите

```
rat p, q(4,5), r=q;
p = q;
...
r = p;
```

При присвояването се копират всички член-данни на обекта. Така присвояването

```
r = p;
```

е еквивалентно на

```
r.numer = p.numer;
r.denom = p.denom;
```

Подробности относно процеса на присвояване са дадени в следващите части на тази глава.

## Някои задачи върху дефиниране на класове

**Задача 118.** Да се дефинира клас "точка в равнината" с две член-данни - двете декартови координати на точката и подходящи член-функции. Като се използва дефинираният клас да се напише програма, която:

а) въвежда n различни точки от равнината, след което ги транслира с (2, 4) и извежда получените точки;

б) намира разстоянието между всеки две точки (все едно старите или новите);

в) намира точките, разстоянието между които е най-малко (най-голямо);

г) проверява, дали въведените точки, в реда, в който са въведени, образуват изпъкнал многоъгълник.

Програма Zad118.cpp решава задачата.

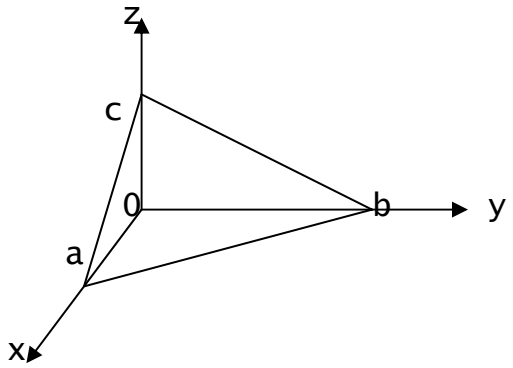
```
// Program Zad118.cpp
#include <iostream.h>
#include <math.h>
class point
{private:
```

```
  double x;
  double y;
 public:
  point(double=0, double=0);
  void read();
  void move(double, double);
  double get_x() const
  {return x;
  }
  double get_y() const
  {return y;
  }
  void print() const;
};
point::point(double a, double b)
{x = a;
 y = b;
}
void point::read()
{cout << "x= ";
 cin >> x;
 cout << "y= ";
 cin >> y;
}
void point::print() const
{cout << "(" << x << ", " << y << ")" << endl;
}
void point::move(double dx, double dy)
{x = x + dx;
 y = y + dy;
}
double dist(point X, point Y)
{return sqrt(pow(Y.get_x()-X.get_x(), 2) +
             pow(Y.get_y()-X.get_y(), 2));
}
int main()
{// a)
 cout << "n= ";
```

```
    int n;
    cin >> n;
    point table[10];
    for (int i=0; i<=n-1; i++)
     table[i].read();
    for (i=0; i<=n-1; i++)
     table[i].print();
    cout << endl;
    for (i=0; i<=n-1; i++)
    {table[i].move(2,4);
     table[i].print();
    } // б)
    for (i=0; i<=n-2; i++)
     for (int j=i+1; j<=n-1; j++)
       cout << dist(table[i], table[j]) << endl;
    return 0;
   }
```

Подточки в) и г) оставяме за самостоятелна работа. Разгледайте добре решението и обяснете ролята на конструктора с подразбиращи се параметри. Какво щеше да стане ако параметрите му не бяха подразбиращи се?

**Задача 119.** Да се дефинират класове “точка в пространството” и “пирамида” с подходящи член-данни и член-функции. Като се използват дефинираните класове да се напише програма, която въвежда n точки в пространството и установява дали всички те принадлежат на пирамидата




включително на контура й.

Програма Zad119. cpp решава задачата.

```
// Program Zad119.cpp
#include <iostream.h>
```

```
class Point
{public:
  void Read();
  double GetX() const;
  double GetY() const;
  double GetZ() const;
 private:
  double x, y, z;
};
void Point::Read()
{cout << "x= "; cin >> x;
 cout << "y= "; cin >> y;
 cout << "z= "; cin >> z;
}
double Point::GetX()const
{return x;
}
double Point::GetY()const
{return y;
}
double Point::GetZ()const
{return z;
}
class Piramid
{public:
  void Read();
  bool IsPointIn(Point const &p) const;
 private:
  double a, b, c;
};
void Piramid::Read()
{cout << "a= "; cin >> a;
 cout << "b= "; cin >> b;
 cout << "c= "; cin >> c;
}
bool Piramid::IsPointIn(Point const &p) const
{double x =  p.GetX()/a + p.GetY()/b + p.GetZ()/c;
 return (p.GetX()>=0 && p.GetY()>=0 && p.GetZ()>=0 &&
```

```
            p.GetX()/a + p.GetY()/b + p.GetZ()/c <=1);
}
int main()
{Piramid p;
 cout << "Input piramid \n";
 p.Read();
 Point pt[100];
 cout << "number of points: ";
 int n; cin >> n;
 for (int i=0; i<=n-1; i++)
 {cout << "Point " << i << ": \n";
  pt[i].Read();
 }
 int x = 0;
 while (x<=n-2 && p.IsPointIn(pt[x])) x++;
 if (p.IsPointIn(pt[x])) cout << "Yes\n";
 else cout <<"No\n";
 return 0;
}
```

## 17.4 Конструктори

Създаването на обекти е свързано с отделяне на памет, запомняне на текущо състояние, задаване на начални стойности и др. дейности, които се наричат **инициализация на обекта**. В езика C++ тези дейности се изпълняват от специален вид член-функции на класовете – конструкторите.

### 14.4.1. Дефиниране на конструктор

На фиг. 14.5 е дадена най-често използваната форма за дефиниране на конструктор.

**Дефиниране на конструктор (най-често използвана форма)**

```
<дефиниция_на_конструктор> ::=
<име_на_клас>::<име_на_клас>(<параметри>)
```

```
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
<параметри> се определя както формални параметри на функция.
```

Фиг. 14.5 Дефиниране на конструктор (най-често използвана форма)

Ще напомним, че конструкторът е член-функция, която притежава повечето характеристики на другите член-функции, но има и редица особености, като:

- Името на конструктора съвпада с името на класа.
- Типът на резултата е указателят this и явно не се указва.
- Изпълнява се автоматично при създаване на обекти.
- Не може да се извиква явно (обръщение от вида r.rat(1,4) е недопустимо).

Освен това в един клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора.

Типична дефиниция на конструктор е дадена в следващия пример.

**Пример:**

```
class CL
{public:
  CL(int, int, int);
  void print();
  ...
 private:
  int a, b, c;
};
CL::CL(int x, int y, int z) // конструктор с три параметъра
{a = x;
 b = y;
 c = z;
}
```

Класът CL в този пример има един триаргументен конструктор. При създаване на обект на класа, този конструктор ще се изпълнява автоматично, стига да е коректно извикан, в резултат на което член-данните a, b и c на създадения обект ще се свържат със стойностите, които се подадат като фактически параметри на конструктора.

На Фиг. 14.6 е дадена по-обща дефиниция на конструктор.

```
Дефиниране на конструктор
<дефиниция_на_конструктор> ::=
<име_на_клас>::<име_на_клас>(<параметри>):
    <член_данна>(<израз>){,<член_данна>(<израз>)}опц
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
```

Фиг. 14.6 Дефиниране на конструктор

Забелязваме, че е възможно член_данна да се свърже с инициализираща стойност в заглавието на конструктора.

**Пример:** Конструкторът на класа CL може да се дефинира и по следния начин

```
CL::CL(int x, int y, int z): a(x), b(y), c(z)
{}
```

Ще отбележим, че не е задължително всички член-данни да са инициализирани само пред тялото или само вътре в тялото на конструктора.

**Пример:** Допустима е дефиницията

```
CL::CL(int x, int y, int z): a(x)
{b = y;
 c = z;
}
```

Смисълът на обобщената синтактична конструкция е, че инициализацията на член-данните в заглавието предшества изпълнението на тялото на конструктора. Това я прави изключително полезна. Използването й увеличава ефективността на класа поради следните съображения. Когато член-данни на клас са обекти, в дефиницията на конструктора на класа при инициализацията се използват конструкторите на класовете, от които са обектите (член-данни). Преди да започне изпълнението на указаните конструктори, автоматично се извикват конструкторите по подразбиране на всички член-данни, които са обекти. Веднага след това тези член-данни се инициализират с обектите, резултат от изпълнението на извиканите конструктори. Това двойно извикване на конструктори намалява ефективността на програмата.

**Пример:** Ще дефинираме класа prat, като член-данна в него е обект на класа rat.

```
class prat
{public:
  prat(int, int, int);
  ...
 private:
  int a;
  rat r;
};
```

където

```
prat::prat(int x, int y, int z)
{a = x;
 r = rat(y, z);
}
```

и сме дефинирали обекта q

```
prat q=prat(1,2,3);
```

Преди да започне изпълнението на конструктора prat, автоматично се извиква конструкторът по подразбиране на rat и член-данната r на prat се инициализира с 0/1. Веднага след това r се свързва с обекта rat(y,z) за текущите y и z. По-ефективно е данната r да се свърже с правилната стойност направо, без междинна инициализация. Това може да се реализира чрез дефиницията:

```
prat::prat(int x, int y, int z): r(rat(y, z))
{a = x;
}
```

или съкратено

```
prat::prat(int x, int y, int z): r(y, z)
{a = x;
}
```

### 14.4.2 Предефинирани конструктори

Обект (в общия смисъл) е предефиниран, ако за него има няколко различни дефиниции, задаващи различни негови интерпретации. За да бъдат използвани такива конструкции е необходим критерии, по който те да се различат.

В рамките на една програма може да се извършва предефиниране на функции. Възможно е:

*а) да се използват функции с едно и също име с различни области на видимост*

В този случай не възниква проблем с различаването.

*б) да се използват функции с едно и също име в една и съща област на видимост*

В този случай компилаторът търси функцията с възможно най-добро съвпадане. Като критерии за добро съвпадане са въведени следните нива на съответствие:

- точно съответствие (по брой и тип на формалните и фактическите параметри)
- съответствие чрез разширяване на типа.
  Извършва се разширяване по веригата
  char -> short -> int -> longint или
  float -> double
- други съответствия (правила въведени от потребителя).

В един клас може да са дефинирани няколко конструктора. Всички те имат едно име (името на класа), но трябва да се различават по броя и/или типа на параметрите си. Наричат се **предефинирани конструктори**. При създаването на обект на класа се изпълнява само един от тях. Определя се съгласно критерия за най-добро съвпадане.

**Пример:** В класа rat дефинирахме два конструктора rat() и rat(int, int), които се различават по броя на параметрите си.

### 14.4.3 Подразбиращ се конструктор

В клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Ако явно не е дефиниран конструктор, автоматично се създава един т. нар. **подразбиращ се конструктор**. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др.

Подразбиращият се конструктор може да бъде предефиниран. За целта е необходимо в класа да бъде дефиниран конструктор без параметри.

**Пример:** В класа rat, подразбиращият се конструктор беше предефиниран от конструктора

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

### 14.4.4 Конструктори с подразбиращи се параметри

Функциите в езика C++ могат да имат подразбиращи се параметри. За тези параметри се задават подразбиращи се стойности, които се използват само ако при извикването на функцията не бъде зададена стойност за съответния параметър.

Задаването на подразбираща се стойност се извършва чрез задаване на конкретна стойност в прототипа на функцията или в нейната дефиниция.

**Пример:** Да разгледаме програмата

```
#include <iostream.h>
void f(double, int=10, char* ="example1"); //интервал между * и
=
int main()
{double x = 1.5;
 int y = 5;
 char z[] = "example 2";
 f(x, y, z);
 f(x, y);
 f(x);
 return 0;
}
void f(double x, int y, char* z)
{cout << "x= " << x << " y= " << y
      << " z= " << z << endl;
}
```

В тази програма е дефинирана функцията f с три формални параметъра. От прототипа й се вижда, че два от тях (вторият и третият) са подразбиращи се със стойности по подразбиране 10 и “example 1” съответно. Тъй като в обръщенията към f

```
f(x, y); и
f(x);
```

са указани по-малко от три фактически параметъра, за стойности на липсващите параметри се вземат указаните стойности от прототипа на функцията.

В резултат от изпълнението на програмата се получава:

```
x = 1.5 y = 5 example 2
x = 1.5 y = 5 example 1
x = 1.5 y = 10 example 1
```

При използване на подразбиращи се параметри, важна роля играе редът на параметрите. Прието е, че ако параметър на функция е подразбиращ се, всички параметри след него също са подразбиращи се.

**Пример:** Прототип на функция

```
void f(double = 1.5, int, char* “example 1”);
```

предизвиква грешка, тъй като първият формален параметър е обявен за подразбиращ се, а вторият не е такъв.

Ще отбележим също, че ако за даден подразбиращ се параметър е зададена стойност при обръщението към функцията, за всички параметри *пред него* също трябва да са указани такива.

Всичко казано за подразбиращите се параметри на функции се отнася и за конструкторите.

Ако променим дефиницията на класа rat по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
```

където конструкторът rat(int, int); се дефинира по същия начин, са допустими следните дефиниции на обекти:

```
rat p,                  // p се инициализира с 0/1
    q = rat(),          // q се инициализира с 0/1
    r = rat(5),         // r се инициализира с 5/1
    s = rat(13,21),     // s се инициализира с 13/21
```

```
t(2);                // t се инициализира с 2/1
```

### 14.4.5 Конструктори за присвояване и копиране

Инициализацията на новосъздаден обект на даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или кръгли скоби.

**Пример:** Нека

```
rat p = rat(1,4);
```

Чрез еквивалентните конструкции

```
rat q = p;
rat q(p);
```

се създава обектът q от клас rat, като инициализацията на q зависи от p. Тази инициализация се създава от специален конструктор, наречен **конструктор за присвояване**.

Конструкторът за присвояване е конструктор, поддържащ формален параметър от тип <име_на_клас> const &.

- Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в момента когато новосъздаден обект се инициализира с обекта, намиращ се от дясната страна на знака за присвояване или в кръглите скоби. Този конструктор за присвояване се нарича конструктор за копиране.

**Пример:** В класа rat не беше дефиниран конструктор за присвояване. Затова при създаване на обект p чрез дефиницията rat p = q;
автоматично се извиква конструкторът за копиране. Последният има вида:

```
rat::rat(rat const & r)
{numer = r.numer;
 denom = r.denom;
}
```

или по-точно

```
rat::rat(rat* this, rat const & r)
{this -> numer = r.numer;
 this -> denom = r.denom;
}
```

Дефиницията rat p=q създава нов обект p *(без викане на конструктор)*, в който се копират съответните стойности на обекта q. Това е резултат от изпълнението на обръщението към rat(&p, q).

- Ако в класа е дефиниран конструктор за присвояване, компилаторът го използва.

**Пример:** Ще добавим един безсмислен, даже глупав, конструктор за присвояване към класа rat. Правим го с експериментална цел.

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(rat const &); // конструктор за присвояване
  rat();
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
...
rat::rat(rat const & r)
{numer = r.numer + 1;
 denom = r.denom + 1;
}
```

Компилаторът превежда този конструктор за присвояване във вида:

```
rat_rat(rat *this, rat const &r)
{this->numer = r.numer + 1;
 this->denom = r.denom + 1;
}
```

а

```
rat q = p;
```

в

```
rat_rat(&q, p);
```

Така дефинираният конструктор за присвояване увеличава с 1 числителя и знаменателя на рационалното число p, фактически параметър на обекта r (формален параметър на конструктора) и ги

свързва с числителя и знаменателя на обекта q сочен от указателя this.

В резултат имаме:

```
rat p,          // p се инициализира с 0/1
    q = p,      // q се инициализира с 1/2
    r = q       // r се инициализира с 2/3
    s = r,      // s се инициализира с 3/4
    t(s);       // t се инициализира с 4/5.
```

*Предефиниране на служебния конструктор за копиране се налага когато обектите имат член-данни, указващи динамична памет.*

Освен в горните случаи, конструкторът за присвояване (копиране) се използва и при предаване на обект като аргумент на функция, когато предаването е по стойност, а също при връщане на обект като резултат от изпълнение на функция.

Правилата, определящи достъпа на функциите до обектите, съществено не се различават от тези, регламентиращи достъпа на функциите до обикновените променливи. Обектите могат да се предават като параметри на функциите по един от известните вече три начина: по стойност, по указател и по псевдоним. При предаване по стойност функциите работят с *копия* на параметрите, а не със самите параметри. При другите два начина за предаване на параметрите не се правят копия (функциите работят с оригиналните параметри).

Когато обектите се предават като параметри на функции, спецификаторите на достъп public и private имат същия смисъл, т.е. функциите имат достъп само до public компонентите на обектите, когато им се подават като параметри.

**Пример:** Нека останем в означенията на класа rat с глупавия конструктор за присвояване от предишния пример и нека функцията sum, намираща сума на две рационални числа, е дефинирана по следния начин:

```
a) rat sum(rat r1, rat r2)
   {rat r(r1.get_numer()*r2.get_denom()+
          r2.get_numer()*r1.get_denom(),
          r1.get_denom()*r2.get_denom());
    return r;
   }
```

Обръщението sum(p, q).print() извежда 8/7. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри стойности, те се свързват с фактическите си параметри чрез присвояване. В резултат r1 се свързва с ½ (не с 0/1), а r2 – с 2/3 (не с ½). След изпълнението на инициализацията

```
rat r(r1.get_numer()*r2.get_denom()+
      r2.get_numer()*r1.get_denom(),
      r1.get_denom()*r2.get_denom());
```

/чрез двуаргументния конструктор rat(int, int)/ обектът r се свързва със 7/6. Тъй като функцията rat е от тип rat, при изпълнение на return r; се прави още едно прилагане на глупавото присвояване и се получава 8/7.

б)
```
rat sum(rat const & r1, rat const & r2)
   {rat r(r1.get_numer()*r2.get_denom()+
          r2.get_numer()*r1.get_denom(),
          r1.get_denom()*r2.get_denom());
    return r;
   }
```

Сега обръщението sum(p, q).print() извежда 2/3. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри псевдоними, те директно се свързват с фактическите си параметри (не се извършва присвояване), т.е. r1 се свързва с 0/1, а r2 – 1/2. След изпълнението на инициализацията

```
rat r(r1.get_numer()*r2.get_denom()+
      r2.get_numer()*r1.get_denom(),
      r1.get_denom()*r2.get_denom());
```

r се свързва със 1/2. Аналогично на случай а), при изпълнение на return r; се прилагане “глупавото” присвояване и се получава 2/3.

в)
```
rat sum(rat* r1, rat* r2)
   {rat r(r1->get_numer()*r2->get_denom()+
         r2->get_numer()*r1->get_denom(),
         r1->get_denom()*r2->get_denom());
   return r;
   }
```

и

```
rat* p1 = &p,
   * q1 = &q;
```

Обръщението sum(p1, q1).print() извежда 2/3. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри указатели, те се свързват с фактическите си параметри чрез адрес. В резултат r1 се свързва с 0/1, а r2 – с 1/2. След изпълнението на инициализацията

```
rat r(r1->get_numer()*r2->get_denom()+
      r2->get_numer()*r1->get_denom(),
      r1->get_denom()*r2->get_denom());
```

r се свързва със 1/2. Тъй като функцията rat е от тип rat, при изпълнение на return r; се прилага присвояването и се получава 2/3.

## 14.5 Указатели към обекти на класове

Дефинират се по същия начин както се дефинират указатели към променливи от стандартните типове данни.

**Пример:**

```
rat p;
rat * ptr = &p;
```

В резултат ще се отделят 4в ОП, които ще се именуват с ptr и ще се инициализират с адреса на обекта p.

Достъпът до компонентите на рационалното число, сочено от ptr, се осъществява по общоприетия начин:

```
(*ptr).get_numer()
(*ptr).get_denom()
```

Синтактичната конструкция (*ptr). е еквивалентна на ptr ->. Така горните обръщения могат да се запишат и по следния начин:

```
ptr -> get_numer()
ptr -> get_denom()
```

Ще напомним, че this е указател от тип <име_на_клас>*.

## 14.6 Масиви и обекти

Елементите на масив могат да са обекти, но разбира се от един и същ клас. Дефинират се по общоприетия начин (Фиг. 14.7).

**Дефиниция на масив от обекти**

<дефиниция_на_променлива_от_тип_масив_от_обекти> ::=

```
      T <променлива>[size] [= {<инициализиращ_списък>}]опц;
```

където

- T е име или декларация на клас;
- <променлива> е идентификатор;
- size е константен израз от интегрален или изброен тип с *положителна* стойност;
- <инициализиращ_списък> се дефинира по следния начин:

```
<инициализиращ_списък> ::= <стойност>{, <стойност>}опц
      {, <име_на_конструктор>(<фактически_параметри>)}опц
```

Фиг. 14.7 Дефиниция на масив от обекти

**Пример:**

```
rat table[10];
```

определя масив от 10 обекта от клас rat.

Достъпът до елементите на масива е пряк и се осъществява по стандартния начин – чрез индексирани променливи.

**Пример:** Чрез индексираните променливи

```
table[0], table[1], ..., table[9]
```

се осъществява достъп до първия, втория и т.н. до десетия елемент на table.

Тъй като table[i] (i = 0, 1, ..., 9) са обекти, възможни са следните обръщения към техни компоненти:

```
table[i].read();        // въвежда стойност на table[i]
table[i].print();       // извежда стойността на table[i]
table[i].get_numer();   // намира числителя на table[i]
table[i].get_denom();   // намира знаменателя на table[i].
```

Връзката между масиви и указатели е в сила и в случая когато елементите на масива са обекти. Името на масива е указател към първия му елемент, т.е. ако

```
rat * p = table;     // p сочи към table[0]
                     // т.е. p==&table[0]
*(p+i) == table[i], i = 0, 1,...,9
```

Тогава

```
(*(p+i)).print();    // е еквивалентно на table[i].print();
```

Масивът може да е член-данна на клас.

**Пример:** Конструкцията

```
class example
```

```
{int a;
 int table[10];
 public:
 int array[10];
} x[5];
```

дефинира масив с 5 компоненти, които са от тип example. Достъпът до компонентите на масива array ще се осъществи по следния начин:

x[i].array[j], i = 0, 1, ..., 4; j = 0, 1, ..., 9.

Конструкторите (в частност конструкторът по подразбиране) играят важна роля при дефинирането и инициализирането на масиви от обекти. Масив от обекти, дефиниран в програма, се инициализира по два начина:

- *неявно* (чрез извикване на системния конструктор по подразбиране за всеки обект – елемент на масива);
- *явно* (чрез инициализиращ списък).

**Примери:**

**а)** Класът

```
const NUM = 5;
class student
{public:
 void read_student();
 void print_student() const;
 bool is_better(student const &) const;
 double average() const;
 private:
 int facnom;
 char name[26];
 double marks[NUM];
};
```

няма явно дефиниран конструктор. Дефиницията

```
student table[30];
```

на масива table от 30 обекта от клас student е правилна. Инициализацията се осъществява чрез извикване на “системния” конструктор по подразбиране за всеки обект – елемент на масива.

б) Класът rat, дефиниран по-долу

```
class rat
{private:
  int numer;
```

```
        int denom;
       public:
        rat(int=0, int=1);
        void read();
        int get_numer() const;
        int get_denom() const;
        void print() const;
      };
```

притежава явно дефиниран конструктор с два подразбиращи се параметъра. В този случай са допустими дефиниции от вида:

```
   rat x[10]; // x[i] се инициализира с 0/1, за всяко i=0,1,...9.
   rat x[10] = {1,2,3,4,5,6,7,8,9,10}; //x[i] == i/1
   rat x[10] = {rat(1,21),rat(2),rat(3, 5),4,5,6,7,8,9,10};
   // x[0] == 1/21; x[1] == 2/1; x[2] == 3/5, x[3]== 4/1, ...
```

Ако променим конструктора на класа rat от

```
   rat(int=0, int=1);
```

в

```
   rat(int, int);
```

т.е. без подразбиращи се параметри и трите дефиниции от по-горе ще съобщят за грешка. Единствено допустима дефиниция на x[10] е с инициализация с 10 обръщения към двуаргументния конструктор rat с явно указани два аргумента.

**Задача 120.** Да се напише програма, която въвежда следната информация за компютри: име на модела (name), цена (price) и точки (score) между 1 и 100. Да се изведе въведената информация, след което да се изведе сортирана в низходящ ред по съотношението точки/цена.

Отново ще използваме подхода абстракция със структури от данни. Първите две нива на абстракция ще реализираме чрез дефиницията на класа

```
   class product
   {public:
     void read();
     void print() const;
     bool is_better_from(product const &) const;
     double get_price() const;
```

```
  int get_score() const;
 private:
  char name[21];
  double price;
  int score;
};
```

Примитивните операции се реализирани чрез следните член-функции:

```
void read();            - въвежда информация за компютър
void print() const;     - извежда информация за компютър
bool is_better_from(product const &) const;
                        -  проверява дали текущият компютър има
                           по-добро съотношение score/price
                           от това на указания като формален
                           параметър
double get_price() const; - намира цената на компютър
int get_score() const;   - намира точките на компютър
```

Програма Zad120.cpp решава задачата.

```
// Program Zad120.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
class product
{private:
  char name[21];
  double price;
  int score;
 public:
  void read();
  void print() const;
  bool is_better_from(product const &) const;
  double get_price() const;
  int get_score() const;
};
void sorttable(int n, product* []);
int main()
{cout << setprecision(4) << setiosflags(ios::fixed);
```

```
 product table[300];
 product* ptable[300];
 int n;
 do
 {cout << "number of products? ";
  cin >> n;
 } while (n<1 || n>300);
 int i;
 for (i = 0; i <= n-1; i++)
 {table[i].read();
  ptable[i] = &table[i];
 }
 cout << "table: \n";
 for (i = 0; i <= n-1; i++)
 {table[i].print();
  cout << endl;
 }
 sorttable(n, ptable);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {ptable[i]->print();
  cout << setw(7)
       << ptable[i]->get_score()/ptable[i]->get_price()
       << endl;
 }
 return 0;
}
void product::read()
{cout << "name: ";
 cin >> name;
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
}
void product::print() const
{cout << setw(25) << name
      << setw(10) << price
```

```
      << setw(12) << score;
}
bool product::is_better_from(product const & x) const
{return score/price > x.score/x.price;
}
double product::get_price() const
{return price;
}
int product::get_score() const
{return score;
}
void sorttable(int n, product* a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  product* max = a[i];
  for (int j = i+1; j <= n-1; j++)
  if (a[j]->is_better_from(*max))
   {max = a[j];
    k = j;
   }
  max = a[i]; a[i] = a[k]; a[k] = max;
 }
}
```

Ще дадем още един пример, показващ връзка между класове и масиви. В него масивът е член-данна на клас, описващ последователно представяне на структурата от данни стек.

## 14.7 Стек

Стекът е линейна динамична структура от данни. В Глава 8 (Увод в програмирането на базата на езика C++) направихме кратко описание на тази структура. В тази част ще направим по-пълно описание. Ще започнем със следната задача.

**Задача 121.** Да се напише програма, която извежда двоичното представяне на естествено число.

Операторът

do

```
{cout << k%2;
 k/=2;
} while (k);
```

извежда двоичното представяне на числото k, но в обратен ред. За решаване на задачата ще използваме динамичната структура от данни **стек**.

Стекът е крайна редица от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими само за единия край на редицата, който се нарича **връх на стека**. Възможен е достъп само до елемента, намиращ се на върха на стека като достъпът е пряк.

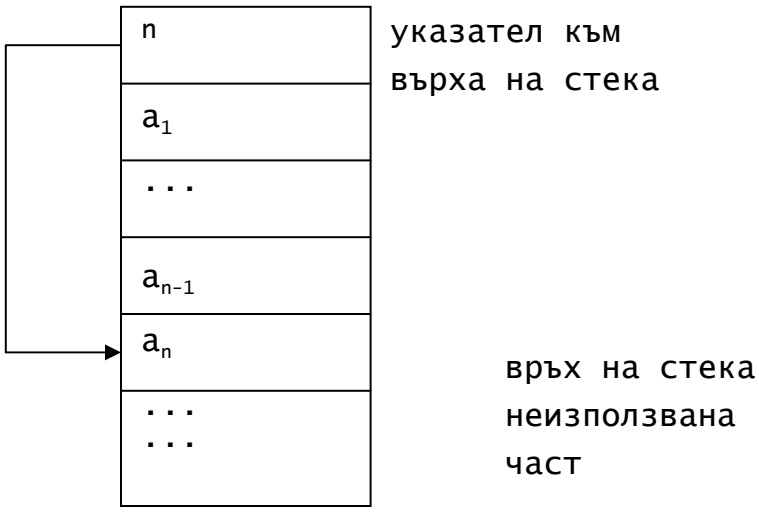
При тази организация на логическите операции, последният включен елемент се изключва пръв. Затова стекът се определя още като структура "*последен влязъл – пръв излязъл*".

Широко се използват два основни начина за физическо представяне (представяне в ОП) на стек: *последователно* и *свързано*. За целите на тази задача ще използваме последователното представяне.

При това представяне се запазва блок от паметта, вътре в който стекът да расте и да се съкращава. Ако редицата от елементи от един и същ тип

$a_n$, $a_{n-1}$, ..., $a_1$

е стек с връх $a_n$, последователното представяне на стека има вида:




При включване на елементи в стека, те се поместват в последователни адреси в неизползваната част веднага след върха на стека.

Това физическо представяне ще реализираме като използваме структурата от дании масив. За указател към върха на стека ще служи цяла променлива. Ще използваме подхода абстракция със структури от данни. Първите две нива на абстракция реализираме чрез класа stack:

```
class stack
{public:
  stack();
  void push(int);
  void pop();
  void print();
  int top() const;
  bool empty() const;
 private:
  int n;    // указател към върха на стека
  int arr[NUM];   // представяне на стека
};
```

където

```
const int NUM = 100;
```

По такъв начин ограничаваме размера на стека до 99 (arr[0] ще инициализираме с 0 и няма да използваме). Масивът arr ще представя стека, а n ще е указателя към върха му.

Примитивните операции са реализирани чрез следните член-функции:

```
void push(int);         - включва елемент в стека
void pop();             - изключва елемент от стека
void print();           - извежда елементите на стека като
                          разрушава стека
int top() const;        - намира елемента от върха на стека
bool empty() const;     - проверява дали стекът е празен
```

Програма Zad121.cpp решава задачата.

```
// Program Zad121.cpp
#include <iostream.h>
const NUM = 100;
class stack
{public:
  stack();
```

```
     void push(int);
     void pop();
     int top() const;
     bool empty() const;
     void print();
    private:
     int n;
     int arr[NUM];
   };
   stack   num_stack(int);   //   конструира   стек   от   двоичното
представяне
                        // на указано цяло число
   void main()
   {cout << "number: ";
    int n;
    cin >> n;
    num_stack(n).print();
   }
   stack::stack()
   {n = 0;
    arr[0]=0;
   }
   void stack::push(int x)
   {n++;
    arr[n] = x;
   }
   void stack::pop()
   {n--;
   }
   int stack::top() const
   {return arr[n];
   }
   bool stack::empty() const
   {return n == 0;
   }
   void stack::print()
   {while (!empty())
    {cout << top();
```

```
    pop();
   }
   cout << endl;
  }
  stack num_stack(int x)
  {stack st;
   while (x)
   {st.push(x%2);
    x/=2;
   }
   return st;
  }
```

## 14.8 Динамични обекти

Вече разгледахме в най-общ план разпределението на ОП по време на изпълнението на програма. От Фиг. 8.1 на Глава 8 се вижда, че всяка програма има две “места” за памет: *програмен стек (стек)* и *област за динамичните данни (динамична памет, хийп или куп).*

***Стекът*** е област за временно съхранение на информация. Той е кратковременна памет. C++ използва стека основно за реализиране на обръщения към функции. Всяко обръщение към функция предизвиква конструиране на нова стекова рамка, която се установява на върха на стека. По такъв начин когато функция A извика функция B, която от своя страна вика функция C, стекът нараства. Когато пък всяка от тези функции завършва, стековите рамки на тези функции автоматично се разрушава. Така стекът се свива.

***Хийпът*** е по-постоянна област за съхранение на данни. Той е един вид дълготрайна памет. Особеност на тази памет е, че тя не се свързва с имена на променливи. С разположените в нея обекти се работи косвено – чрез указатели. Обикновено се използва при работа с т. нар. **динамични структури от данни**. *Динамичните данни са такива обекти (в широкия смисъл на думата), чийто брой не е известен в момента на проектирането на програмата. Те се създават и разрушават по време на изпълнението на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва отново. Така паметта се използва по-ефективно.*

Използването на динамичната памет досега не се налагаше, тъй като структурите от данни, с които работехме, бяха статични. За целите на следващите разглеждания, когато ще дефинираме и използваме динамичните структури от данни свързан списък, стек, опашка, дърво, граф и др., използването на тези средства е задължително.

Създаването и разрушаването на динамични обекти в C++ се осъществява чрез операторите new и delete. Извикването на new заделя в хийпа необходимата памет и връща указател към нея. Този указател може да се съхрани в някаква променлива и да се пази докато е необходимо. За разлика от стека, заделянето на памет в хийпа е явно – чрез new. Освобождаването на паметта от хийпа също става явно, чрез delete. Всяко извикване на new трябва да бъде балансирано чрез извикване на delete. Последното се налага, тъй като за разлика от стека, хийпът не се изчиства автоматично. В C++ няма система за "събиране на боклуци" (автоматично премахване на обекти, които вече не са необходими). Затова трябва явно да бъдат изтрити създадените в хийпа обекти. Описанието на оператора new е дадено на Фиг. 14.8.

**Оператор new**

*Синтаксис*

new <име_на_тип> [ [size] ]$_{опц}$;|

new <име_на_тип> (<инициализация>);

където

- *<име_на_тип>* е име на някой от стандартните типове int, double, char и др. или е име на клас;

- *size* е израз с произволна сложност, но трябва да може да се преобразува до цял. Показва броя на компонентите от тип <име_на_тип>, за които да се задели памет в хийпа и се нарича **размерност**;

- *<инициализация>* е израз от тип <име_на_тип> или инициализация на обект според синтаксиса на конструктора на класа, ако <име_на_тип> е име на клас.

*Семантика*

Заделя в хийпа (ако е възможно):

- sizeof(<име_на_тип>) байта, ако не са зададени size и <инициализация> или

- sizeof(<име_на_тип>)*size байта, ако явно е указан size или

- sizeof(<име_на_тип>) байта, ако е специфицирана <инициализация>, която памет се инициализира с <инициализация> и връща указател към заделената памет.

Ако няма достатъчно памет в хийпа, операторът new връща NULL.

Фиг. 14.8 Оператор new

**Забележка:** Ако <име_на_тип> е име на клас и след него има кръгли скоби, в тях трябва да стоят фактически параметри (аргументи) на конструктора на класа. Ако скобите липсват, класът трябва да притежава конструктор по подразбиране или да няма явно дефиниран конструктор. Ако след името на класа е поставен заграден в квадратни скоби израз, new заделя място за масив от обекти на указания клас и извиква конструктора по подразбиране за инициализиране на отделената памет.

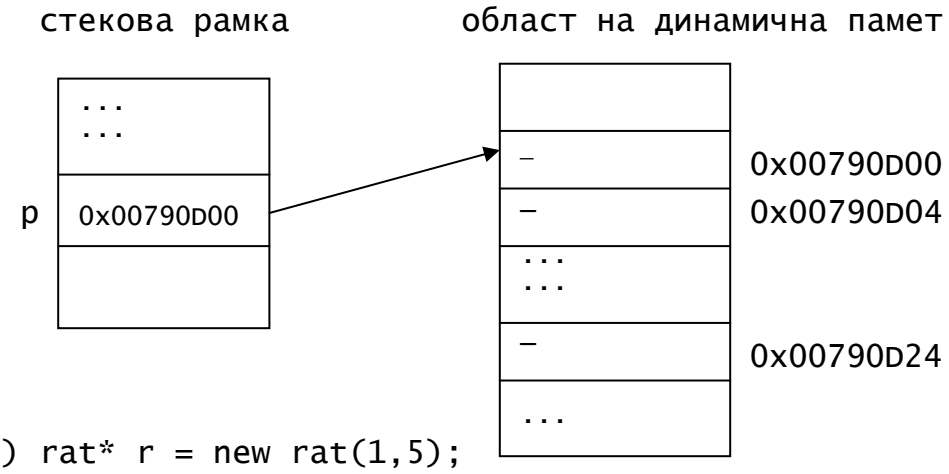
**Примери:**

а) int* q = new int(2+5*5);

отделя (ако е възможно) 4B памет в хийпа, инициализира я с 27 – стойността на израза 2+5*5 и свързва q с адреса на тази памет, т.е.

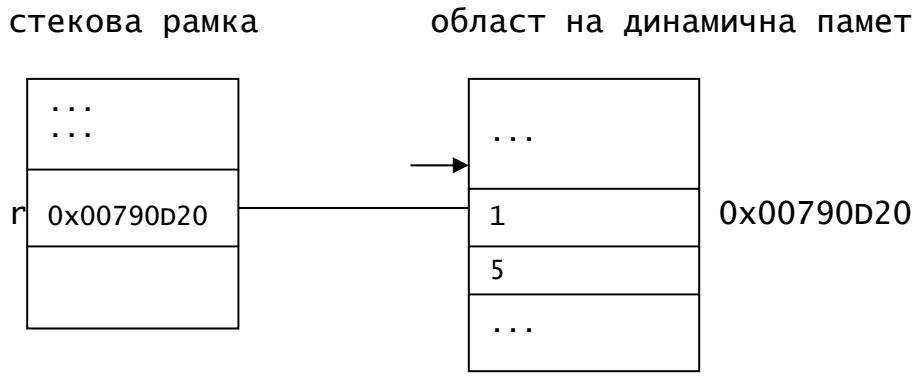
б) int* p = new int[10];

отделя (ако е възможно) 40B в хийпа (за 10 елемента от тип int) и свързва p с адреса на тази памет, т.е.



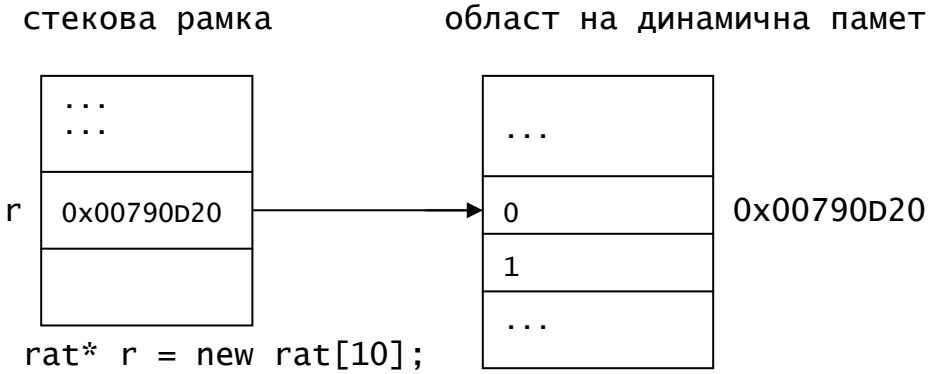

в) rat* r = new rat(1,5);

отделя памет в хийпа за един обект от клас rat, свързва r с адреса на тази памет и извиква конструктора rat(1,5) за да я инициализира, т.е.




г) rat* r = new rat;

отделя памет в хийпа за обект от тип rat, записва адреса на тази памет в r и извиква конструктора по подразбиране на класа rat за инициализиране на тази памет, т.е.

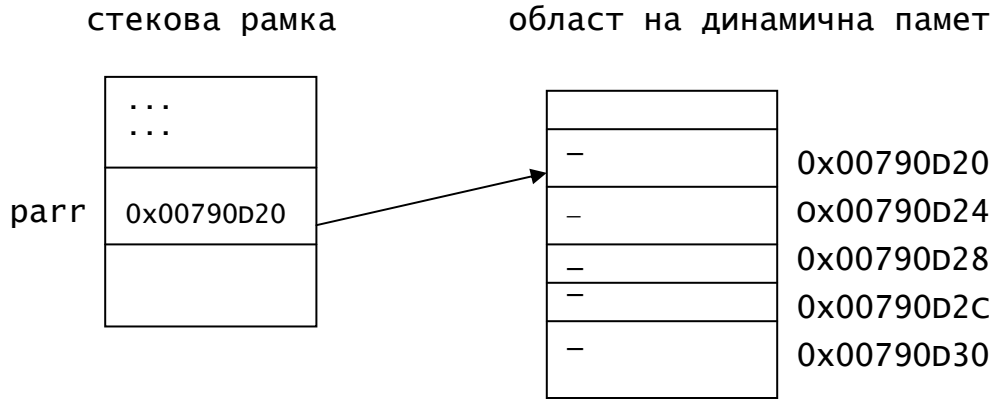



д) rat* r = new rat[10];

отделя памет в хийпа за 10 обекта от класа rat, записва адреса на тази памет в r, извиква конструктора по подразбиране на класа rat и инициализира отделената памет;

е) rat** parr = new rat*[5];

отделя 20в памет в хийпа за масив от 5 указателя към стойности от тип rat и записва в parr адреса на тази памет, т.е.



Заделянето на памет по време на компилация се нарича **статично** заделяне на памет, заделянето на памет по време на изпълнение на програмата – **динамично разпределение на паметта**. Паметта за променливите q, p, r и parr, от примерите по-горе, е заделена статично, а всяка една от тези променливи има за стойност адрес от хийпа. Казва се още, че q, p, r и parr адресират динамична памет.

Под период на **активност на една променлива** се разбира частта от времето за изпълнение на програмата, през което променливата е свързана с конкретно място в паметта. Паметта за глобалните променливи се заделя в началото и остава свързана с тях до завършването на изпълнението на програмата. Паметта за локалните променливи се заделя при влизане в локалната област и се освобождава при напускането ѝ. Паметта на динамичните променливи се заделя от оператора new. Заделената по този начин памет остава свързана със съответната променлива докато не се освободи явно от програмиста. Явното освобождаване на динамична променлива се осъществява чрез оператора delete, приложен към указателя, който адресира съответната променлива. Едно непълно негово описание е дадено на Фиг. 14.9.

**Оператор delete**

*Синтаксис*

delete <указател_към_динамичен_обект>;

където <указател_към_динамичен_обект> е указател към динамичен обект (в широкия смисъл на думата), създаден чрез оператора new.

*Семантика*

Разрушава обекта, адресиран от указателя, като паметта, която заема този обект, се освобождава. Ако обектът, адресиран от указателя, е обект на клас, отначало се извиква деструкторът на класа (т.15.9) и след това се освобождава паметта.

Фиг. 14.9 Оператор delete

Ако в хийпа е заделена памет, след което тази памет не е освободена чрез delete, се получава загуба на памет. Парчето памет, което не е освободено, е като остров в хийпа, заемащо пространство, което иначе би могло да се използва за други цели.

За да се разруши масив, създаден чрез new по следния начин:

```
int* arr = new int[5];
```

трябва да се запише:

```
delete [] arr;
```

**Забележка:** Някои реализации на езика допускат разрушаването в горния случай да стане и чрез delete arr.

Ако обаче масивът съдържа в себе си указатели, първо трябва да бъде обходен и да бъде извикан операторът delete за всеки негов елемент. Едва след това може да се извика delete за масива по показания по-горе начин.

Следващата задача илюстрира казаното.

**Задача 122.** Да се напише програма, която отделя динамична памет за масив от 10 указателя към тип int. Програмата да проверява дали отделянето на памет е станало; запълва масива с указатели към цели числа; извежда стойностите и съдържанието на указателите и освобождава отделената динамична памет.

```
// Program Zad122.cpp
#include <iostream.h>
int main()
{// заделяне на памет за масив от указатели към int
 int**  arr = new int*[10];
 if (arr == NULL)  //или if (!arr)
 {cout << "Not enough memory!\n";
  return 1;
 }
 // запълване на масива с указатели
 int i;
 for (i=0; i<10; i++)
 {arr[i] = new int;     // заделяне на памет за указателя
  if (arr[i]==NULL) //или if (!arr[i])
   {cout << “Not enough memory!\n”;
    return 1;}
  *arr[i] = i;    // инициализиране на указваната стойност
 }
 // извеждане на стойностите и съдържанието на указателите
 for (i=0; i<10; i++)
```

```
      cout << arr[i] << “  “ << *arr[i] << ", ";
    cout << endl;
    // освобождаване на заетата динамична памет
    for (i = 0; i < 10; i++)
      delete arr[i];
    delete [] arr;
    return 0;
   }
```

Операторът delete трябва да се използва само за освобождаване на динамична памет, заделена с new. В противен случай действието му е непредсказуемо. Няма забрана за прилагане на delete към указател със стойност 0. Ако стойността на указателя е 0, той е свободен и не адресира нищо.

**Пример:**

```
   void example()
   {...
    int a = 7;
    char *str = “abv”;
    int *pa = &a;
    rat *ptr = 0;
    double *x = new double;
    delete str; //некоректно обръщение, str не е адресирано чрез new
    delete pa; //некоректно обръщение, pa не е адресирано чрез new
    delete ptr; //некоректно обръщение, ptr не е адресирано чрез new
    delete x; // коректно обръщение
    ...
   }
```

Динамичната памет не е неограничена. Тя може да се изчерпи по време на изпълнение на програмата. Неуспешното завършване на delete ускорява изчерпването. Ако наличната в момента динамична памет е недостатъчна, new връща нулев указател и програмата (функцията) няма да работи. Затова се препоръчва след всяко извикване на new да се прави проверка за успешността ѝ.

Чрез оператора new могат да се създават т. нар. **динамични масиви** – масиви с променлива дължина. Динамичните масиви се

създават в динамичната памет. Следващата програма илюстрира този процес.

**Задача 123.** Да се напише програма, която създава динамичен масив от цели числа. Да се изведе масивът.

```
// Program 123. cpp
#include <iostream.h>
int main()
{int size; // дължина на масива
 do
 {cout << "size of array: ";
  cin >> size;
 } while (size<1);
 // създаване на динамичен масив arr от size елемента от тип int
 int* arr = new int[size];
 int i;
 for (i = 0; i <= size-1; i++)
  arr[i] = i;
 // извеждане на елементите на arr
 for (i = 0; i <= size-1; i++)
  cout << arr[i] << " ";
 cout << endl;
 // освобождаване на заетата динамична памет
 delete [] arr;
 return 0;
}
```

Чрез оператора

```
delete [] arr;
```

е разрушен динамичният масив arr. Това може да стане и ако се напише само

```
delete arr;
```

тъй като елементите на масива arr са числа.

Забелязваме, че размерът size на динамичния масив може да се въведе или изчисли по време на изпълнение на програмата и не е задължително да бъде известен по време на компилация. Това

позволява да се подобри програмата на задача 120, като се използват динамични, а не “статични” масиви (т. 14.8).

Методите на класовете също могат да използват динамична памет, която се заделя и освобождава по време на изпълнението им, чрез операторите new и delete.

**Задача 124.** Да се модифицира класът product, реализиран в задача 120, така че за всяко име на компютър да се отделя точно толкова памет, колкото е необходимо, а не точно 21 байта.

За целта ще определим променливата name като указател към char и необходимата памет за съхраняването на името на компютър ще се заделя по време на изпълнение на член-функцията read(). Следват само фрагментите, където се налага модификация.

```
char s[40];
class product
{private:
  char* name;
  double price;
  int score;
 public:
  void read();
  void print() const;
  bool is_better_from(product const &) const;
  double get_price() const;
  int get_score() const;
};
...
void product::read()
{cout << "name: ";
 cin >> s;
 name = new char[strlen(s)+1];
 strcpy(name, s);
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
}
```

Използвана е глобална променлива s от тип низ, която играе ролята на буфер – временно съхранява въведено име. След това се намира дължината на този низ и в динамичната памет за name се отделя точно толкова памет, колкото е необходимо. Така за член-данната name на всеки обект на класа product ще се заделя нужната памет, а не винаги 21 B, която памет може да се окаже и недостатъчна.

Ще отбележим също, че заделената от член-функциите динамична памет не се освобождава автоматично при разрушаване на обектите на класове. Освобождаването на тази памет трябва да стане явно чрез оператора delete, който трябва да се изпълни преди разрушаването на обекта. Този процес може да бъде автоматизиран чрез използване на специален вид методи, наречени **деструктори**.

## 14.9 Деструктори

Разрушаването на обекти на класове в някои случаи е свързано с извършване на определени действия, които се наричат **заключителни**. Най-често тези действия са свързани с освобождаване на заделена преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност заключителните действия да се извършат автоматично при разрушаването на обекта. Това се осъществява от деструкторите.

Деструкторът е член-функция, която се извиква при:
- разрушаването на обект чрез оператора delete,
- излизане от блок, в който е бил създаден обект от класа.

Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа, предшествано от символа '~' (тилда), типът му е void и явно не се задава в заглавието. Деструкторът няма формални параметри.

**Забележка:** Използването на явно дефинирани деструктори не винаги е належащо, тъй като всички член-променливи се разрушават при разрушаването на обекта и без използването на деструктор. Ако конструкторът или някоя член-функция реализира динамично заделяне на памет за някоя член-данна, използването на деструктор е задължително, тъй като в този случай той трябва да освободи заетата памет.

**Задача 125.** Да се промени класът product, дефиниран в програма Zad120.cpp, като методът read() се преобразува в конструктор по подразбиране и се добави деструктор. Освен това table и ptable да се реализират като динамични масиви.

Програма Zad125.cpp решава тази задача. Освен исканите модификации тя добавя и функцията за достъп

char* get_name() const;

която не е използвана в това приложение. Телата на някои методи и функции на програмата, които не са модифицирани, са пропуснати.

```
// Program Zad125.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
char s[40];
class product
{private:
  char* name;
  double price;
  int score;
 public:
  product();
  ~product();
  void print() const;
  bool is_better_from(product const &) const;
  char* get_name() const;
  double get_price() const;
  int get_score() const;
};
void sorttable(int n, product* a[]);
int main()
{cout << setprecision(4) << setiosflags(ios::fixed);
 cout << "size: "; // размерност на масива
 int size;
 cin >> size;
 //създава динамичен масив от size обекта на product
 product* table = new product[size];
```

```
 // заделя памет за динамичен масив от указатели
 // към size обекта на product
 product** ptable = new product*[size];
 int i;
 cout << "table: \n";
 for (i = 0; i <= size-1; i++)
 {table[i].print();
  cout << endl;
  ptable[i] = &table[i];
 }
 sorttable(size, ptable);
 cout << "\n New Table: \n";
 for (i = 0; i <= size-1; i++)
 {ptable[i]->print();
  cout << setw(7)
       << ptable[i]->get_score()/ptable[i]->get_price()
       << endl;
 }
 delete [size] table; // някои реализации допускат
                      // пропускането на size
 delete [] ptable;      // някои реализации допускат
                        // пропускането на []
 return 0;
}
product::~product()
{delete name;
}
product::product()
{cout << "name: ";
 cin >> s;
 name = new char[strlen(s)+1];
 strcpy(name, s);
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
}
void product::print() const
```

```
{cout << setw(25) << name
      << setw(10) << price
      << setw(12) << score;
}
bool product::is_better_from(product const & x) const
{return score/price > x.score/x.price;
}
char* product::get_name() const
{return name;
}
double product::get_price() const
{return price;
}
int product::get_score() const
{return score;
}
void sorttable(int n, product* a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  product* max = a[i];
  for (int j = i+1; j <= n-1; j++)
  if (a[j]->is_better_from(*max))
   {max = a[j];
    k = j;
   }
  max = a[i]; a[i] = a[k]; a[k] = max;
 }
}
```

В резултат от изпълнението на оператора

```
product* table = new product[size];
```

се създава динамичен масив table със size обекта на класа product. При създаването на всеки от тези обекти се изпълнява конструкторът на класа, т.е. за всеки обект на table ще бъдат въведени име, цена и точки.

Операторът

```
product** ptable = new product*[size];
```

предизвиква заделяне на памет за динамичен масив от указатели към size обекта на product.

Накрая, чрез операторите

```
delete [size] table;
delete [] ptable;
```

се разрушават динамичните масиви. При разрушаването на масива table деструкторът се изпълнява size пъти, което води до освобождаване на заделената чрез конструктора памет за съхраняване на имената на обектите.

Недостатък на горната програма е, че не анализира резултата от new. Възможно е да няма достатъчно памет в хийпа. Тази ситуация е критична и изпълнението на програмата трябва да завърши. Това може да се коригира като след използването на new се добавят програмни фрагменти от вида:

```
if (!table)
{cout << "Not enough memory!\n";
 return 1;
}
```

**Забележка:** Ако се освобождава памет, заета от динамичен масив, чийто елементи са обекти на клас, трябва явно да се посочи дължината на масива. Тя е необходима за да се определи броят на извикванията на деструктора.

По повод на това, че за всяко обръщение към new трябва да има съответен delete, възниква въпросът: *Когато функция върне указател или псевдоним към обект, създаден чрез new, кой носи отговорността за извикването на delete за този указател?*

Например, къде в програмния фрагмент

```
struct object
{int a, b;
}
object& myfunc();
int main()
{object& rmyobj = myfunc();
 cout << rmyobj.a << rmyobj.b << endl;
 return 0;
}
object& myfunc()
{object *o = new object;
 o->a = 20;
 o->b = 40;
```

```
 return *o; // връща самия обект
}
```

да бъде изтрит rmyobj?

Функцията, която създава указателя или псевдонима нищо не може да направи, защото когато указателят или псевдонимът бъде върнат, тя вече ще е завършила. Така че, който е извикал функцията, той след това трябва да извика и delete. Възможно е извикващият да е програма, принадлежаща на друг програмист или ваша стара програма и да не помните тази подробност. Затова като цяло *се смята за лошо програмиране връщането на указател, който после трябва да бъде изтрит. По-добре е да се върне обекта по стойност.* В примерната програма по-горе в main, след извеждането на rmyobj трябва да се включи операторът delete &rmyobj.

## 14.10 Създаване и разрушаване на обекти на класове

Съществуват два начина за създаване на обекти:

- чрез дефиниция;
- чрез функциите за динамично управление на паметта.

**В първия случай** обектът се създава при срещане на дефиницията (във функция или блок) и се разрушава при завършване на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори. Дефиницията, чрез която се създава обект, може да бъде допълнена с инициализация, която може да се основава на извикване на обикновен конструктор или на конструктора за присвояване.

Разрушаването на обекта е свързано с извикването на деструктора на класа, ако такъв явно е дефиниран или с извикването на “системния” деструктор (деструктора по подразбиране), ако в класа явно не е дефиниран деструктор.

**Пример:** Нека в класа rat добавим деструктора

```
rat::~rat()
{cout << “destruct number: “
      << number << “/”
      << denum << endl;
}
```

Този избор на деструктор направихме с цел да наблюдаваме къде той ще бъде извикан. Ще напомним, че деструкторът извършва заключителни действия, определени от програмиста (или компилатора). Нека сега изпълним програмата с класа rat, в който е включен и деструкторът ~rat() с главна функция от вида:

```
void main()
{rat p(1,8);   // създава обект p и го инициализира с 1/8
 rat q=rat(2,9);  // създава обект q и го инициализира с 2/9
 for(int i=1; i<=5; i++)
 {rat r(i, i+1);  //  създава  обект  r  и  го  инициализира  с
i/(i+1)
  r.print();      // за i = 1, ..., 5
 }
 p.print();
 q.print();
}
```

В резултат от изпълнението на main се получава:

```
1/2
destruct number:1/2
2/3
destruct number:2/3
3/4
destruct number:3/4
4/5
destruct number:4/5
5/6
destruct number:5/6
1/8
2/9
destruct number:2/9
destruct number:1/8
```

От изпълнението се вижда, че деструкторът на класа rat е извикан толкова пъти, колкото  пъти са извършвани дейности по създаване на обекти на класа rat. Пътвите пет извиквания на деструктора са при завършване на изпълнението на блока на оператора for, а последните две – при завършване на изпълнението на функцията main.

**Във втория случай** създаването и разрушаването на обекти се управлява от програмиста. Създаването става с new, а разрушаването

чрез delete. Операциите new се включват в конструкторите, а операциите delete – в деструктора на класа.

**Пример:**

```
rat *p = new rat(3,7); // търси в хийпа 8B, свързва адреса
                       // им с p, извиква конструктора
                       // rat(9,13) и инициализира паметта
(*p).print();
delete p;            // извиква деструктора, след което
                     // разрушава обекта
...
```

В този случай деструкторът само регистрира присъствието си. Получаваме:

```
3/7
destruct number: 3/7
```

## 14.11 Инициализиране на обекти на класове

Езикът C++ позволява обектите на класове (както и обикновените променливи) да бъдат инициализирани при дефиницията си и при извикването на функцията new. При обикновените променливи инициализаторът задава стойност на променливата, а при обектите – осигурява аргументи на конструкторите. Инициализацията на обект на клас се извършва по следните начини:

```
<име_на-клас> <обект>(<инициализатор>);|
<име_на-клас> <обект> = <инициализатор>;
```

Възможни са:

*а) инициализаторът не е обект на класа*

В този случай се създава обекта, след това се намира стойността на израза–инициализатор и се подава на подходящия конструктор (ако има такъв).

**Пример:** Нека в класа rat конструкторът е с прототип:

```
rat(int = 0; int = 1);
```

Инициализацията

```
rat p = 7;
```

ще създаде обекта p и ще извика конструктора rat(7), с който ще инициализира p. Ако в класа rat не беше дефиниран конструктор с един аргумент, опитът за тази инициализация щеше да е неуспешен.

*б) инициализаторът е обект на класа*

В този случай съществуват някои особености. Ако съществува явно дефиниран конструктор за присвояване, той се използва. В противен случай се използва подразбиращия се конструктор за копиране.

*Конструктор за присвояване явно не е дефиниран*

Тогава се извиква подразбиращия се системен конструктор за копиране.

**Пример:**

```
rat p(1,9);
rat q = p;
```

Създава се нов обект q с член-данни абсолютни копия на съответните член-данни на p.

В този случай възникват проблеми ако някоя член-данна на обекта е указател към динамичната памет, тъй като член-променливата на новия обект, който е указател към динамичната памет, е със същата стойност като на стария (указва към същата памет). В този случай става поделяне на компонента на обектите.

**Пример:** ще използваме класа product, като ще направим някои модификации в него:

```
class product
{private:
  char* name;
  double price;
  int score;
 public:
  ~product();
  product();
  void print() const;
  bool is_better_from(product const &) const;
  char* get_name() const;
  double get_price() const;
  int get_score() const;
};
product::~product()
{delete name;
 cout << "destruct data for: " << this << endl;
}
```

```
product::product()
{cout << "name: ";
 cin >> s;
 name = new char[strlen(s)+1];
 strcpy(name, s);
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
 cout << "new data: " << this << endl;
}
...
void main()
{product p;
 product q = p;
}
```

В този случай дефинираният конструктор по подразбиране product() се извиква веднъж – при инициализирането на p. Тъй като няма явно дефиниран конструктор за присвояване, генерираният от системата конструктор за копиране откопирва член-данните на обекта p в q като член-данната name е поделена. При завършване на блока – тяло на main, първо се разрушава обектът q. За него се извиква деструкторът. Поделената памет се освобождава, след което се разрушава и q. Забележете, q е разрушен, но е разрушена и част от p - поделената динамична памет. После започва процедурата по разрушаването и на обекта p. Извиква се деструкторът, който се опитва да освободи вече освободена памет. Това води до грешка, чийто последствия са непредвидими.

*Конструктор за присвояване явно е дефиниран*

Вече дефинирахме един глупав конструктор за присвояване за класа rat и правихме експерименти с него. Ще напомним, че конструкторът за присвояване е член-функция от вида:

```
<име_на_клас>(<име_на_клас> const&)
{<тяло>}
```

Ще дефинираме подходящ конструктор за присвояване в класа product и ще го извикаме за да реализираме коректно инициализацията от последния пример.

```
product::product(product const & p)
```

```
{name = new char[strlen(p.name)+1];
 strcpy(name, p.name);
 price = p.price;
 score = p.score;
}
```

и включваме прототипа му

```
product(product const & p);
```

в public-частта на класа product.

Тогава функцията

```
void main()
{product p;
 product q = p;
}
```

вече работи добре. Дефинирани са два обекта p – инициализиран чрез конструктора по подразбиране product() и q – чрез дефинирания явно конструктор за присвояване. Деструкторът е извикан два пъти при завършване изпълнението на блока и разрушава обектите q и p.

Дефинираният конструктор за присвояване решава проблемите, възникващи при инициализацията на обект на клас product. Използва се при предаване на обект по стойност, а също при връщане на обект като стойност на функция. Като цяло обаче той не решава проблемите на операцията за присвояване.

**Пример:** Ако променим main по следния начин:

```
void main()
{product p, q;
 q = p;
}
```

отново възникват проблеми. Присвояването q = p; ще промени член-данните на q, но q вече има част в динамичната памет, която *първо трябва да бъде освободена*. Налага се да бъде дефинирана нова операторна функция за присвояване. Последното ще направим по следния начин:

```
product& product::operator=(product const & p)
{cout << "assignment!\n";
 if(this != &p)
 {delete name;
  name = new char[strlen(p.name)+1];
  strcpy(name, p.name);
```

```
     price = p.price;
     score = p.score;
    }
    return *this;
   }
```

като ще включим прототипа й

```
   product& operator=(product const &);
```

в public-частта на класа.

Забелязваме, че операторната функция за присвояване извършва аналогични действия на тези на конструктора за присвояване. Разликата е, че тя извършва тези действия върху съществуващ обект, а не върху токущо създаден. Това налага предварително да бъде освободена динамичната памет, отделена за обекта.

Дефинираната операторна функция има за формален параметър константен псевдоним от клас product. По този начин се избягва създаването на нов обект и извикването на конструктора за присвояване. Въпреки, че не е задължително, използването на константен псевдоним е препоръчително. Това позволява на компилатора да следи за евентуална промяна на обекта – фактически параметър. Освен, че променя обекта, указван от this, операторната функция от примера връща като резултат псевдонима му. Като следствие, конструкцията p = q може да се разглежда като израз (p = q връща p), а също да бъде лява страна на израз. Изразът p = q = r е допустим и е еквивалентен на q = r; p = q;

В този пример реализирахме като член-функции на класа product конструктор за присвояване, операторна функция за присвояване и деструктор. Тези три функции се наричат “голямата тройка” или “канонична форма на класа”. На пратика те са задължителни при класове, използващи динамичната памет. Това не е просто добра идея, това е закон.

## 14.12 Масиви от обекти

Създаването на масив от обекти става по два начина:

- чрез дефиниция
- чрез функциите за динамично управление на паметта.

**При първия начин** масивът от обекти се създава при срещането на дефиницията (във функция или блок) и се разрушава при завършване

на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори.

**Примери:** Ще използваме класа rat

```
a) {...
    rat x[10];
   }
```

В този случай конструкторът rat() е извикан 10 пъти. Конструиран е масивът от обекти x:

```
x[0]  x[1]  ...   x[9]
0/1   0/1   ...   0/1
```

При завършване изпълнението на блока деструкторът ~rat() ще бъде извикан също 10 пъти за да разруши последователно x[9], x[8], ..., x[0].

```
б) {rat x[10] = {rat(1,2), rat(5), 8, rat(1,7)};
   }
```

В този случай конструкторът rat(int = 0, int = 1) е извикан 10 пъти. Конструиран е масивът от обекти x:

```
x[0]  x[1]  x[3]  x[4]  x[5]  x[6]  ...   x[9]
1/2   5/1   8/1   1/7   0/1   0/1   ...   0/1
```

При завършване изпълнението на блока деструкторът ~rat() ще бъде извикан също 10 пъти за да разруши последователно x[9], x[8], ... x[0].

**При втория начин**, създаването и разрушаването на масив от обекти се управлява от програмиста. Отново създаването става чрез new, а разрушаването – с delete, като операторите new се включват в конструкторите, а операторите delete – в деструкторът на класа, от който са обектите на масива.

**Примери:**

```
a){rat *px = new rat[10];
   delete[] px;
  }
```

В резултат, в хийпа се заделя блок от 80 байта (ако е възможно) и адресът на този блок се записва в px. Тъй като има дефиниран конструктор по подразбиране, конструкторът се извиква и px[i] (i=0, 1, ..., 9) се инициализират с рационалното число 0/1. При масивите, реализирани в динамичната памет, инициализация в явен

вид не може да се зададе. Разрушаването на px става чрез delete[] px;. Преди да прекъсне връзката между px и динамичната памет, операторът delete[] извиква деструктора за всеки от обектите на масива.

б) {rat *px = new rat[10];
delete px;
}

В резултат, в хийпа се заделя блок от 80 байта (ако е възможно) и адресът на блока се записва в px. Тъй като в класа има конструктор по подразбиране, този конструктор се извиква и px[i] (i=0, 1, ..., 9) се инициализират с рационалното число 0/1. Операторът delete px; извиква деструктора само на обекта px[0] и прекъсва връзката на px с динамичната памет. Компилаторът съобщава за грешка. Причината е, че px е масив от обекти в динамичната памет, а деструкторът на класа rat е извикан само за px[0]. Ще отбележим отново, че ако px беше масив в динамичната памет, но не от обекти на клас, операторът delete px; щеше да работи нормално.

в) {int size;
cin >> size;
rat* px = new rat[size];
delete[size] px;
}

В този случай деструкторът на класа rat се извиква size пъти. Реализацията на Visual C++ 6.0 пренебрегва size от delete, но за някои други реализации това не е така.

## 14.13 Приятелски класове и функции

Често е необходимо съвместното използване на два класа. Обектно-ориентираното програмиране налага капсолирането на данните. Достъпът до private-компонентите на даден клас от функция извън класа е забранено. В редица случаи това е сериозно затруднение. Например, дефинирани са два класа, представящи вектор и матрица съответно. Функцията, която ще реализира произведението на вектор с матрица ще трябва да има достъп до членовете и на двата класа. Един начин за реализирането на това е да се направят член-данните и на двата класа public. Това ще доведе до загубване на предимствата на капсолирането. Друг начин е да се използват

public функции на достъп, осъществяващи достъп до стойностите на член-променливите. Това води до забавяне на изпълнението на програмата. Трети начин за решаване на този проблем е декларирането на функции или класове – приятели на класа. Приятелите на даден клас (функции или класове) имат достъп до всички негови компоненти, т.е. членовете на класа са винаги public за функциите приятели. Ако клас е декларираи като приятел, всички негови член-функции стават функции приятели.

Примери за функции и класове приятели ще дадем в следващите части на тази глава и книгата.

## 14.14 Оператори. Предефиниране на оператори

Езикът C++ има богат набор от оператори. В него са дадени също средства за предефиниране на оператори.

Всеки оператор се характеризира с:

- позиция на оператора спрямо аргументите му;
- приоритет;
- асоциативност.

**Позицията** на оператора спрямо аргументите му го определя като: префиксен (операторът е пред единствения му аргумент), инфиксен (операторът е между аргументите си) и постфиксен (операторът е след аргумента си).

**Пример:** Операторът / е инфиксен (4/8), операторът + е както инфиксен, така и префиксен (2+8, +78), а операторът ++ е както постфиксен, така и префиксен.

**Приоритетът** определя реда на изпълнение на операторите в операторен терм. Оператор с по-висок приоритет се изпълнява преди оператор с по-нисък приоритет.

**Пример:** Приоритетът на * и / е по-висок от този на + и -.

**Асоциативността** определя реда на изпълнение на оператори с еднакъв приоритет в операторен терм. В C++ има лявоасоциативни и дясноасоциативни оператори. Лявоасоциативните оператори се изпълняват отляво надясно, а дясноасоциативните – отдясно наляво.

В C++ не могат да се дефинират нови оператори, но всеки съществуващ едноаргументен или двуаргументен оператор с изключение на ::, ?:, ., *, # и ## може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас.

Например, възможно е да се предефинират операторите +, -, * и /, така че да могат да събират, изваждат, умножават и делят рационални числа. Тогава вместо sum(p, q), sub(p, q), mult(p, q) и quot(p, q) ще можем да пишем p+q, p-q, p*q и p/q, което безспорно е много по-удобно.

Предефинирането се осъществява чрез дефиниране на специален вид функции, наречени **операторни функции**. Последните имат синтаксис като на обикновените функции, но името им се състои от запазената дума operator, следвана от мнемоничното означение на предефинирания оператор. Когато предефинирането на оператор изисква достъп до компонентите на класове, обявени като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел на тези класове. Предефинираният оператор запазва всички характеристики на оригиналния.

Предефинирането може да стане по два начина:

- чрез функция-приятел;
- чрез член-функция.

Чрез примери ще покажем тези два начина.

**Предефиниране чрез функция-приятел**

**Задача 126**. Да се предефинират операторите +, -, * и / така, че да могат да бъдат използвани за събиране, изваждане, умножение и деление на рационални числа.

Програма Zad126_1.cpp решава задачата. В public частта на класа rat са включени декларациите на предефинираните оператори, предшествани от запазената дума friend:

```
friend rat operator+(rat const &, rat const &);
friend rat operator-(rat const &, rat const &);
friend rat operator*(rat const &, rat const &);
friend rat operator/(rat const &, rat const &);
```

а след дефиницията на функцията main са дадени и техните дефиниции.

```
// Program Zad126_1.cpp
#include <iostream.h>
class rat
{private:
```

```
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  ~rat()
  {cout << "destruct number: " << numer <<
         "/" << denom << endl;
  }
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
  friend rat operator+(rat const &, rat const &);
  friend rat operator-(rat const &, rat const &);
  friend rat operator*(rat const &, rat const &);
  friend rat operator/(rat const &, rat const &);
};
rat::rat(int a, int b)
{numer = a;
 denom = b;
 cout << "construct! \n";
}
void rat::read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 } while (denom == 0);
}
int rat::get_numer() const
...
int rat::get_denom() const
...
void rat::print() const
...
int main()
{rat p(1,3), q(2,5), r(p+q);
```

```
 r.print();
 r = p-q-q;
 r.print();
 return 0;
}
// предефиниране на оператора +
rat operator+(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.denom +
       r2.numer*r1.denom,
       r1.denom*r2.denom);
 return r;
}
// предефиниране на оператора -
rat operator-(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.denom-
       r2.numer*r1.denom,
       r1.denom*r2.denom);
 return r;
}
// предефиниране на оператора *
rat operator*(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.numer,
       r1.denom*r2.denom);
 return r;
}
// предефиниране на оператора /
rat operator/(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.denom,
       r1.denom*r2.numer);
 return r;
}
```

**Забележки:**

1) Изразът p+q се интерпретира като извикване на операторната функция operator+(p, q).
2) Запазва се асоциативността. Изразът p-q-r се интерпретира като (p-q)-r.

**Предефиниране чрез член-функция**

В този случай първият аргумент на член-функцията трябва да е обект на класа и при дефинирането на операторната функция не се задава като параметър. Ако това не е така, операцията не може да се предефинира като член-функция.

Ще модифицираме предишната програма като дефинираме операторните функции за събиране, изваждане, умножение и деление като член-функции на класа rat.

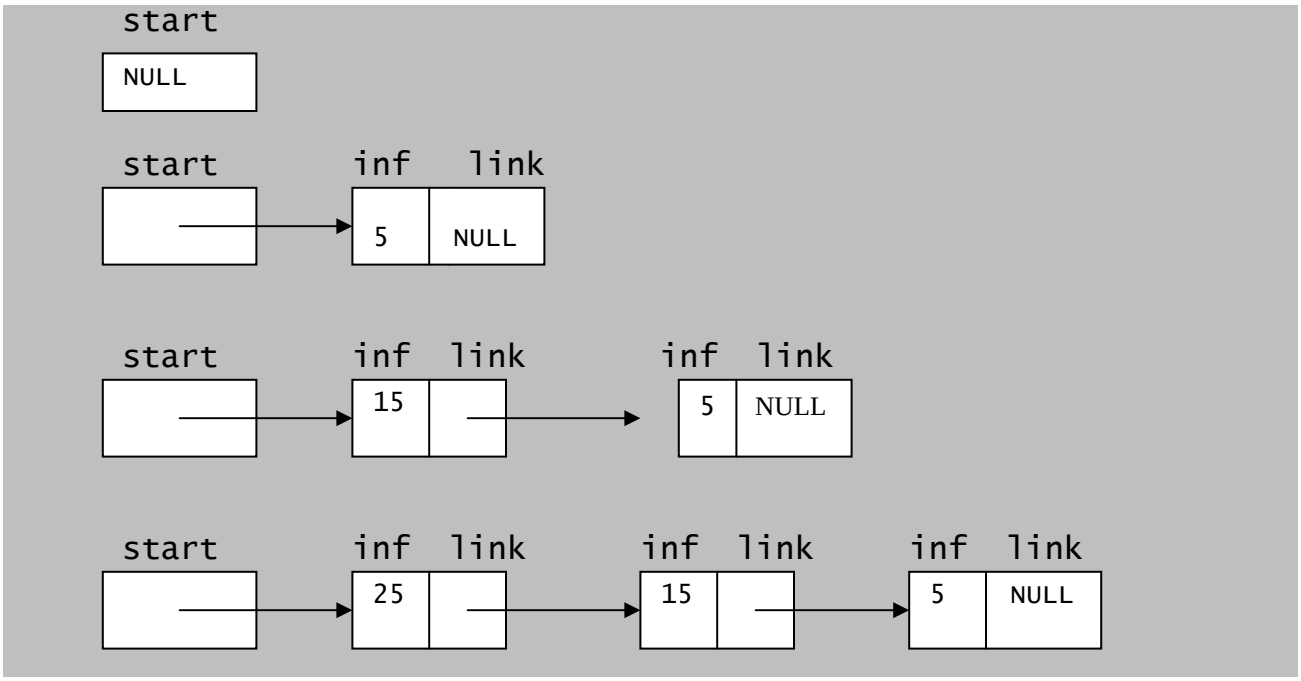
```
// Program Zad126_2.cpp
#include <iostream.h>
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  ~rat()
  {cout << "destruct number: " << numer
      << "/" << denom << endl;
  }
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
  rat operator+(rat const &) const;
  rat operator-(rat const &) const;
  rat operator*(rat const &) const;
  rat operator/(rat const &) const;

};
rat::rat(int a, int b)
...
void rat::read()
...
int rat::get_numer() const
...
int rat::get_denom() const
```

```
...
void rat::print() const
...
int main()
{rat p(1,3), q(2,5), r(p+q);
 r.print();
 r = p-q-q;
 r.print();
 return 0;
}
rat rat::operator+(rat const & r1) const
{rat r(numer*r1.denom + r1.numer*denom,
       denom*r1.denom);
 return r;
}
rat rat::operator-(rat const & r1) const
{rat r(numer*r1.denom - denom*r1.numer,
       denom*r1.denom);
 return r;
}
rat rat::operator*(rat const & r1) const
{rat r(numer*r1.numer,
       denom*r1.denom);
 return r;
}
rat rat::operator/(rat const & r1) const
{rat r(numer*r1.denom,
       denom*r1.numer);
 return r;
}
```

Ще отбележим, че в този случай изразът p+q се интерпретира като p.operator+(q).

## 14.15 Приложение на средствата за работа с динамичната памет

Ще конструираме клас stack, който ще реализира свързаното представяне на стек от цели числа. Фиг. 14.10 илюстрира това представяне.




Фиг. 14.10 Свързано представяне на стек

Забелязваме, че има указател start, който в първия случай представя празен стек, а в останалите случай – непразен, като сочи двойна кутия с информационна част (inf) от тип int и свързваща част (link) от типа на start. Това представяне ще реализираме по следния начин:

```
struct elem
{int inf;
 elem* link;
} *start = NULL, *p;
```

След тази дефиниция start представя празен стек. Включването на елемента 5 можем да направим чрез изпълнение на следните действия:

```
p = start;
start = new elem;
start->inf = 5;
start->link = p;
```

Включването на 15 ще направим по аналогичен начин

```
p = start;
start = new elem;
start->inf = 15;
start->link = p;
```

а на 25 – чрез

```
p = start;
start = new elem;
start->inf = 25;
start->link = p;
```

Тези разсъждения показват, че който и да е елемент x може да се **включи** в стека чрез изпълнение на фрагмента:

```
p = start;
start = new elem;
start->inf = x;
start->link = p;
```

**Изключването** на елемент от последния стек от фиг. 14.10 води до получаване на стека, илюстриран на по-горната стъпка на същата фигура и може да се реализира така:

```
p = start;
x = start->inf;
start = start->link;
delete p;
```

В x е запомнен изключеният елемент. Така получаваме следната непълна реализиция на свързаното представяне на стек от цели числа:

```
struct elem
{int inf;
 elem* link;
};
class stack
{public:
  stack(); // конструктор по подразбиране
  void push(int const&); // член-функция за включване на елемент
  int pop(int & x);  // член-функция за изключване на елемент
 private:
  elem *start;
};
stack::stack()
{start = NULL;
}
void stack::push(int const& s)
```

```
{elem* p = start;
 start = new elem;
 start->inf = s;
 start->link = p;
}
int stack::pop(int & s)
{elem *p;
 if (start)
 {s = start->inf;
  p = start;
  start = start->link;
  delete p;
  return 1;
 }
return 0;
}
```

Забелязваме, че pop връща 1 ако операцията изключване е възможна и 0 – ако не е.

Тъй като обектите на класа stack са реализирани в динамичната памет, за него трябва да реализираме каноничното представяне.

*деструктор*

Единствената член-данна на класа stack е указател към динамичната памет, където е разположен стекът. Затова деструкторът трябва да изтрие стека от паметта. Тъй като изтриването на стек ще е необходимо и за други цели, ще го реализираме чрез член-функцията delstack() на класа.

```
void stack::delstack()
{elem *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
```

Тогава деструкторът ~stack() има вида:

```
stack::~stack()
{delstack();
}
```

*конструктор за присвояване*

Конструкторът за присвояване

```
stack(stack const & r);
```

откопирва стека r в неявния параметър и ще го реализираме по следния начин:

```
stack::stack(stack const & r)
{copy(r);
}
```

където copy(r) ще дефинираме като член-функция на stack и тя ще реализира копиране на стека r в неявния параметър.

*операторна функция за присвояване*

Ще я реализираме така:

```
stack& stack::operator=(stack const& r)
{if (this != &r)
 {delstack();
  copy(r);
 }
 return *this;
}
```

Копирането ще реализираме чрез член-функцията copy(stack const &r). За целта ще сканираме елементите на r (без да ги разрушаваме) и ще ги запишем на друго място в динамичната памет.

```
void stack::copy(stack const & r)
{if (r.start) // r не е празен
{elem *p = r.start, *q = NULL, *s=NULL;
 start = new elem;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
 while (p)
 {s = new elem;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
```

```
   }
   else start = NULL;
  }
```

Класът stack, реализиращ свързаното представяне на стек, има вида:

```
   struct elem
   {int inf;
    elem* link;
   };
   class stack
   {public:
     stack();
     ~stack();
     stack(stack const &);
     stack& operator=(stack const & r);
     void push(int const&);
     int  pop(int & x);
     bool empty() const;
     void print();
    private:
     elem *start;
     void delstack();
     void copy(stack const&);
   };
   stack::stack()
   {start = NULL;
   }
   stack::~stack()
   {delstack();
   }
   stack::stack(stack const & r)
   {copy(r);
   }
   stack& stack::operator=(stack const& r)
   {if (this != &r)
   {delstack();
    copy(r);
   }
   return *this;
```

```
}
void stack::delstack()
{elem *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
void stack::copy(stack const & r)
{if(r.start)
{elem *p = r.start, *q = NULL, *s=NULL;
 start = new elem;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
 while (p)
 {s = new elem;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
}
else start = NULL;
}
void stack::push(int const& s)
{elem* p = start;
 start = new elem;
 start->inf = s;
 start->link = p;
}
int stack::pop(int & s)
{if (start)
 {s = start->inf;
  elem *p= start;
```

```
     start = start->link;
     delete p;
     return 1;
    }
    else return 0;
   }
   void stack::print()
   {int x;
    while (pop(x))
      cout << x << " ";
    cout << endl;
   }
   bool stack::empty() const
   {return start==NULL;
   }
```

В него са добавени и член-функциите:

```
   void print();
```

която извежда елементите на стек и

```
   bool empty() const;
```

проверяваща дали стек е празен.

Следващите програми използват този клас.

**Задача 127.** Да се напише функция void sortstack(stack s, stack & ns), която сортира елементите на стека от цели числа s по метода на пряката селекция. Стекът ns съдържа резултата.

Функцията

```
   void minstack(stack s, int& min, stack &newst);
```

намира минималния елемент на стека s, а също стека newst, съдържащ елементите на s без минималния.

```
   void minstack(stack s, int& min, stack &newst)
   {int x;
    s.pop(min);
    while (s.pop(x))
    if (x<min)
     {newst.push(min);
      min = x;
     }
```

```
 else newst.push(x);
 }

void sortstack(stack s, stack& ns)
{int min;
 while (!s.empty())
 {stack s1;
  minstack(s, min, s1);
  ns.push(min);
  s = s1;
 }
}
```

Едно приложение на структурата от данни стек е пресмятането на стойности на изрази.

**Задача 128.** От клавиатурата е въведена записана без грешка формула от вида:

```
<формула> ::=  <цифра>|
               M(<формула>, <формула>)|
               m(<формула>, <формула>)
<цифра> ::= 0|1|...|9,
```

където M означава функция за намиране на максимума на две цифри, а m – функция за намиране на минимума на две цифри. Въвеждането продължава до срещане на символа ‘.’. Например, стойността на формулата 8 е 8, а стойността на M(1,m(9,6)) е 6.

Функцията formula решава задачата. Тя използва помощен стек от символите ‘M’, ‘m’ и цифрите и реализира следния алгоритъм. Ако прочетеният символ е ‘M’, ‘m’, ‘0’, ...,’9’, се записва в стека s. Ако е ‘)’, от s се изключват три елемента. Тъй като въвежданата формула е правилна (по условие) изключените елементи са два операнда и операция (M - max или m - min). Извършва се операцията и полученият резултат се включва в стека. Останалите символи (интервали, ‘(‘, ‘,’, символите за преминаване на нов ред) се пропускат.

Помощният стек от символи ще реализираме като обект на *клас stack от символи*, който получаваме като заместим в дефиницията на класа stack типа int с char.

```
int formula()
{char c, x, y, op;
 stack st;
 cin >> c;
 while (c!='.')
 {if (c=='m'||c=='M'||c>='0'&&c<='9') st.push(c);
  else
     if (c==')')
     {st.pop(y);
      st.pop(x);
      st.pop(op);
      switch (op)
      {case 'm': if(x<y)c=x;else c=y; break;
       case 'M': if(x>y)c=x;else c=y;
      }
      st.push(c);
     }
     cin >> c;
 }
 st.pop(c);
 return (int)c-(int)'0';
}
```

## 14.16 Шаблони на функции и класове

В предните разглеждания създадохме класа stack, реализиращ стек от цели числа. След това за други цели се наложи да променим този клас в клас, реализиращ стек от символи. Промяната беше елементарна – просто заменихме типа int на елементите на стека с char. Възможно е обаче в рамките на една и съща програма да е необходимо конструирането на няколко стека от различен тип данни. Така възниква необходимостта от средства, реализиращи *класове, зависещи от параметри*, задаващи типове данни и при конкретни приложения параметрите да се конкретизират.

Такива средства са **шаблоните**. Те позволяват създаването на класове, използващи неопределени (хипотетични) типове данни за своите аргументи и по такъв начин позволяват да бъдат описвани “обобщени” типове данни. Използват се за изграждане на общоцелеви

класове-контейнери (стекове, опашки, списъци и др.). Например, чрез шаблон може да се дефинира обобщен клас за стек с неопределен тип на елементите, след което от шаблона да се получат специфични класове (клас, реализиращ стек от реални числа; клас, реализиращ стек от символи и т.н.). При наличие на шаблони на класове възниква необходимостта и от шаблони на функции, използващи шаблони на класове. Например, искаме на реализираме сортиране на елементите на шаблон на стек. Налага се да дефинираме шаблони на функциите за намиране на минимален елемент на стек с произволен тип на елементите и сортиране на елементите на стек с произволен тип на елементите.

### 14.16.1 Шаблони на функции

Дефиницията на шаблон на функция е дадена на Фиг. 14.11.

**Дефиниция на шаблон на функция**

<дефиниция_на_шаблон_на_функция> ::=

template < class <параметър> {,class <параметър>}$_{опц}$ >

<тип_на_функция> <име_на_функция>(<формални_параметри>)

<тяло>

където

- <параметър> и <име_на_функция> са идентификатори;
- <формални_параметри> и <тяло> се определят както при дефиниция на функция. В тях са използвани указаните параметри на шаблона, вместо конкретните типове.

Фиг. 14.11 Дефиниция на шаблон на функция

Дефиницията започва със запазената дума template (шаблон), следвана от списък от параметри на шаблона, които трябва да участват като типове на аргументи на дефинираната като шаблон функция.

Използването на дефинираните шаблони на функции става чрез обръщение към “обобщената” функция, която шаблонът дефинира, с параметри от конкретен тип. Компилаторът генерира т. нар. **шаблонна функция**, като замества параметрите на шаблона с типовете на

съответните фактически параметри. *При това заместване не се извършват преобразувания на типове.*

**Задача 129.** Да се напише програма, която дефинира шаблон на процедура за въвеждане елементите на масив и шаблон на функция, намираща минималния елемент на масив от елементи, които могат да се сравняват.

```
// Procedure Zad129.cpp
#include <iostream.h>
template <class T>
void read(int n, T* a)
{for (int i = 0; i<=n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
}
template <class T>
T minarray(int n, T* a)
{T min = a[0];
 for (int i = 1; i<=n-1; i++)
  if (a[i]<min) min = a[i];
 return min;
}
int main()
{cout << "n: ";
 int n;
 cin >> n;
 int a[10];
 read(n, a);
 cout << minarray(n, a) << endl;
 double b[10];
 read(n, b);
 cout << minarray(n, b) << endl;
 return 0;
}
```

### 14.16.2 Шаблони на класове

Дефинирането на шаблон на клас се състои от декларация на шаблона и дефиниране на член-функциите му. На Фиг. 14.12 е даден непълно синтаксисът на декларацията на шаблон на клас.

**Декларация на шаблон на клас**

<декларация_на_шаблон_на_клас> ::=

template < class <параметър>[=<име_на_тип>]$_{опц}$

{,class <параметър>[=<име_на_тип>]$_{опц}$}$_{опц}$

>

class <име_на_шаблон_на_клас>

<тяло>;

където

- <параметър>, <име_на_шаблон_на_клас> и <име_на_тип> са идентификатори. В <тяло> са използвани указаните параметри на шаблона, вместо конкретните типове.

Фиг. 14.12 Декларация на шаблон на клас

Броят на параметрите на шаблон на клас е произволен. Параметрите могат да участват на произволни места в дефиницията на шаблона. Освен това е възможно някои или всички параметри на шаблона да са подразбиращи се. Това се осъществява чрез добавяне на инициализацията =<име_на_тип> след името на параметъра. В този случай, при пропускане на параметър, се използва подразбиращият се тип.

**Пример:** Декларацията

```
template <class T, class S = int> class CLASS
{public:
 T func1(T x, S y);
 S func2(T x, S y);
 private:
  T a;
  S b;
};
```

определя шаблон на клас CLASS с два параметъра T и S, като вторият е подразбиращ се – при неуказване, S се интерпретира като тип int.

Дефинирането на член-функции на шаблон се осъществява по два начина – като вградени и не като вградени (описани извън декларацията). При дефинирането на вградените член-функции няма особености. Ако е необходимо, използват се параметрите на класа.

**Пример:**

```
template <class T, class S = int> class CLASS
{public:
  T func1(T x, S y)     // вградена член-функция
  {cout << "func1 \n";
   return x;
  }
  S func2(T x, S y);
 private:
  T a;
  S b;
 };
```

В другия случай дефиницията се предшества от

template <списък_от_параметри>

а пълното име на член-функцията на шаблона се получава с префикса

<име_на_шаблон_на_клас> **<** <параметър> {,<параметър>}$_{опц}$ **>**

**Пример:**

```
template <class T, class S>
S CLASS<T, S>::func2(T x, S y)
{cout << "func2 \n";
 return y;
}
```

**Забележка:** Префиксът се използва и когато член-функция на шаблона е от тип шаблон на клас, указател или псевдоним на шаблон на клас.

Шаблоните на класове не са истински класове, а описания, които се използват от компилатора за създаване на конкретни (шаблонни) класове. Наричат се още **специализации на шаблона на класа**.

Фиг. 14.13 дава дефиницията на шаблонни класове.

**Дефиниция на шаблонен клас**

<дефиниция_на_шаблонен_клас> ::=

```
            <име_на_шаблон_на_клас> < <тип>, <тип>,... >
   <тип> ::= <име_на_тип>
```

Фиг. 14.13 Дефиниция на шаблонен клас

Ако някой <тип> е пропуснат, използва се подразбиращият се, ако декларацията на шаблона е с подразбиращи се параметри, или се съобщава за грешка. При срещане на декларация на шаблонен клас, на базата на зададените типове, компилаторът генерира съответен шаблонен клас.

**Пример:**

а) typedef CLASS<int, double> obj1;

дефинира класа obj1, който е специализация на шаблона на класа CLASS при T – int и S – double. Дефиницията

obj1 o1;

определя o1 за обект на класа obj1, който може да се обръща към public-членовете на obj1, т.е. допустимо е обръщението

o1.func1(5, 10.87);

б) typedef CLASS<int> obj2;

дефинира класа obj2, който е специализация на шаблона на класа CLASS при T – int и S – int. Дефиницията

obj2 o2;

определя o2 за обект на класа obj2, който може да се обръща към public-членовете на obj2, т.е. допустимо е обръщението

o2.func2(5,8);

**Забележка:** Ако на CLASS и двата параметър бяха подразбиращи се, щеше да е възможна и специализацията:

typedef CLASS <> obj3; // ъгловите скоби <> трябва да присъстват

Като илюстрация на казаното ще дефинираме и използваме шаблон на класа stack.

**Задача 130.** Да се дефинират шаблон на класа stack и шаблонна функция за сортиране на елементите на шаблон на стек.

Програма Zad130.cpp решава задачата.

// Program Zad130.cpp

```
#include <iostream.h>
template <class T>
struct elem
{T inf;
 elem* link;
};
template <class T>
class stack
{public:
  stack();
  ~stack();
  stack(stack const &);
  stack& operator=(stack const & r);
  void push(T const&);
  int  pop(T & x);
  bool empty() const;
  void print();
 private:
  elem<T> *start; // указател към шаблона на структурата elem
  void delstack();
  void copy(stack const&);
};
template <class T>
stack<T>::stack()
{start = NULL;
}
template <class T>
stack<T>::~stack()
{delstack();
}
template <class T>
stack<T>::stack(stack<T> const & r)
{copy(r);
}
template <class T>
stack<T>& stack<T>::operator=(stack<T> const& r)
{if (this != &r)
{delstack();
```

```
 copy(r);
}
return *this;
}
template <class T>
void stack<T>::delstack()
{elem<T> *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
template <class T>
void stack<T>::copy(stack<T> const & r)
{if (r.start)
{elem<T> *p = r.start, *q = NULL, *s=NULL;
 start = new elem<T>;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
 while (p)
 {s = new elem<T>;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
}
else start = NULL;
}
template <class T>
void stack<T>::push(T const& s)
{elem<T> *p = start;
 start = new elem<T>;
 start->inf = s;
```

```
 start->link = p;
}
template <class T>
int stack<T>::pop(T & s)
{if (start)
 {s = start->inf;
  elem<T> *p= start;
  start = start->link;
  delete p;
  return 1;
 }
 else return 0;
}
template <class T>
void stack<T>::print()
{T x;
 while (pop(x))
   cout << x << " ";
 cout << endl;
}
template <class T>
bool stack<T>::empty() const
{return start == NULL;
}
template <class T>
void minstack(stack<T> s, T& min, stack<T> &newst)
{T x;
 s.pop(min);
 while (s.pop(x))
 if (x<min)
  {newst.push(min);
   min = x;
  }
 else newst.push(x);
}
template <class T>
void sortstack(stack<T> s, stack<T>& ns)
{T min;
```

```
    while (!s.empty())
      {stack<T> s1;
       minstack(s, min, s1);
       ns.push(min);
       s = s1;
      }
   }
   void main()
   {typedef stack<int> IntStack;
    IntStack s, s1;
    s.push(10); s.push(1); s.push(5); s.push(12);
    s.push(8); s.push(14); s.push(19); s.push(0);
    sortstack(s, s1);
    s1.print();
   }
```

Във функцията main чрез

```
   typedef stack<int> IntStack;
```

е дефиниран класът IntStack. Освен това са дефинирани два шаблона на функции - за намиране на минимален елемент на стек с елементи от тип T и за сортиране на елементи на стек от тип T.

Шаблонът на клас дефинира съвкупност от класове. Понякога е по-удобно за някакъв тип данни член-функция на шаблона на класа да се реализира по различен алгоритъм. В C++ е възможно предефинирането на член-функция на шаблон на клас за конкретен тип. Този процес се нарича **специализация на член-функция**.

**Пример:** Член-функцията print() на шаблона на stack извежда елементите на стек от тип T. Ако стекът, за който я използваме е от символи – ще бъдат изведени символите. Ако е необходимо само в този случай да се извеждат ASCII-кодовете на символите, може да бъде предефинирана член-функцията print() като самостоятелна функция от вида:

```
   void stack<char>::print()
   {char x;
    while (pop(x))
      cout << (int)x << " ";
    cout << endl;
   }
```

Последната ще бъде използвана само при обръщение за извеждане на стек от символи, т.е.

```
typedef stack<char> CharStack;   // дефиниция на клас CharStack
CharStack s;         // s е обект на класа CharStack
s.push('u'); s.push('b');
s.push('p'); s.push('x');
s.print();  // ще се изведат ASCII-кодовете
            // на символите на стека.
```

Шаблоните на класове могат да имат за *приятели* класове, функции, шаблони на класове и шаблони на функции.

**Пример:** Нека имаме дефинициите:

```
class CLASS1 {...};
double func1(...){...}
template <class T> class CLASS2{...};
template <class T> double func2(T){..};
```

В дефиницията на шаблон на нов клас може да се въведат декларациите:

```
friend class CLASS1;
friend double func1(...);
friend class CLASS2<int>;  // само специализацията за int на
                           // шаблона на класа CLASS2 е приятел
friend double func2(double);  // само специализацията за
                              // double е приятел
friend class CLASS2<T>;       // всеки обект на шаблона
                              // на класа CLASS2 е приятел
friend double func2(T);       // всяка функция, създадена
                              // от шаблона, е приятел.
```

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. К. Хорстман, Принципи на програмирането със C++, София, ИК СОФТЕХ, 2000.
3. Ал. Стивънс, Кл. Уолнъм, C++ БИБЛИЯ, София, АЛЕКССОФТ, 2000.
4. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.

# 15

# Линейни динамични структури от данни. Стек. Опашка. Свързан списък. Приложения

## 15.1 Стек

В предишната глава разгледахме логическото описание и физическото представяне на структурата от данни стек. Реализирахме както последователното, така и свързаното представяне на стек. При реализацията на свързаното представяне дефинирахме двойната кутия

| inf | link |
|---|---|

чрез структура.

В тази част ще направим още една реализация на стек. Ще предложим също някои негови приложения.

### 15.1.1 Още една реализация на свързаното представяне на стек

Ще дефинираме класа от цели числа stack, който използва помощния и малко странен клас item за реализиране на двойната кутия.

```
class item
{friend class stack;
 private:
  item(int x = 0)    // конструктор
  {inf = x;
```

```
      link = NULL;
     }
     int inf;
     item* link;
   };
```

Тъй като item използва класа stack в декларацията си, прототипът на класа stack трябва да предшества декларацията на item. Странността на item произтича от това, че всичките му членове са капсолирани. Чрез декларацията

```
   friend class stack;
```

item позволява само класът stack да създава и обработва негови обекти. Конструкторът на item има един подразбиращ се аргумент, което позволява да бъде използван и като конструктор по подразбиране.

Класът stack има вида:

```
   class stack
   {public:
     stack();
     stack(int x);
     ~stack();
     stack(stack const &);
     stack& operator=(stack const &);
     int push(int const&);
     int pop(int & x);
     int top() const;
     bool empty() const;
     void print();
    private:
     item *start;
     void delstack();
     void copy(stack const&);
   };
```

Предложени са два конструктора:

```
   stack::stack()
   {start = NULL;
   }
```

и

```
stack::stack(int x)
{start = new item(x);
}
```

Необходими са за да могат да бъдат създавани празен стек и стек с един (указан като аргумент) елемент.

Тъй като обектите на класа stack са реализирани в динамичната памет, за него трябва да реализираме каноничното представяне – деструктор, конструктор за присвояване и операторна функция за присвояване. Тези член-функции са аналогични на съответните от реализацията на стека при предишното представяне.

*деструктор*

```
stack::~stack()
{delstack();
}
```

*конструктор за присвояване*

```
stack::stack(stack const & r)
{copy(r);
}
```

*операторна функция за присвояване*

```
stack& stack::operator=(stack const& r)
{if (this != &r)
{delstack();
 copy(r);
}
return *this;
}
```

където член-функциите delstack() и copy(stack const &) също са аналогични на тези от предишното представяне.

Класът stack реализира стек от цели числа. В следващото приложение ще имаме нужда от два класа стек: клас стек от цели числа и клас стек от символи. Затова, като използваме проектирания клас stack, ще дефинираме шаблон на клас stack за горното представяне.

```
template <class T>
class stack;
template <class T>
```

```
class item
{friend class stack<T>;
 private:
  item(T x = 0)
  {inf = x;
   link = 0;
  }
  T inf;
  item* link;
};
template <class T>
class stack
{public:
  stack(T x);
  stack();
  ~stack();
  stack(stack const &);
  stack& operator=(stack const &);
  void push(T const&);
  int pop(T & x);
  T top() const;
  bool empty() const;
  void print();
 private:
  item<T> *start;
  void delstack();
  void copy(stack const&);
};
template <class T>
stack<T>::stack(T x)
{start = new item<T>(x);
}
template <class T>
stack<T>::stack()
{start = NULL;
}
```

```
template <class T>
stack<T>::~stack()
{delstack();
}
template <class T>
stack<T>::stack(stack<T> const & r)
{copy(r);
}
template <class T>
stack<T>& stack<T>::operator=(stack<T> const& r)
{if (this != &r)
{delstack();
 copy(r);
}
return *this;
}
template <class T>
void stack<T>::delstack()
{item<T> *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
template <class T>
void stack<T>::copy(stack<T> const & r)
{if (r.start)
{item<T> *p = r.start,
         *q = NULL,
         *s = NULL;
 start = new item<T>;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
```

```
 while (p)
 {s = new item<T>;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
}
else start = NULL;
}
template <class T>
void stack<T>::push(T const& x)
{item<T> *p = new item<T>(x);
 p->link = start;
 start = p;
}
template <class T>
int stack<T>::pop(T & x)
{item<T> *temp,
         *p = start;
 if (p)
 {x = p->inf;
  temp = p;
  p = p->link;
  delete temp;
  start = p;
  return 1;
 }
 else return 0;
}
template <class T>
T stack<T>::top() const
{return (*start).inf;
}
template <class T>
```

```
void stack<T>::print()
{T x;
 while (pop(x))
   cout << x << " ";
 cout << endl;
}
template <class T>
bool stack<T>::empty() const
{return start == NULL;
}
```

Ще използваме тази дефиниция на шаблона stack за да покажем как става пресмятането на стойността на аритметичен израз чрез преобразуването му в обратен полски запис.

### 15.1.2 Приложение на структурата от данни стек за пресмятане на стойност на аритметичен израз

Ще разглеждаме аритметични изрази от вида:

```
<израз> ::= <терм>|
            <израз>+<терм>|
            <израз>-<терм>
<терм> ::= <множител>|
            <терм>*<множител>|
            <терм>/<множител>
<множител> ::= <цифра>|<променлива>|(<израз>)|
              <множител> ^ <цифра>
<цифра> ::= 0|1| ... |9
<променлива> ::= <буква>
```

където ^ означава операцията степенуване.

Пресмятането на стойността на аритметичен израз се реализира по-просто, ако изразът е записан не в обичайния му инфиксен вид, а в т. нар. **обратен полски запис**. Характерно за този вид запис на израз е, че знакът за операция се записва след аргументите си.

**Примери:**

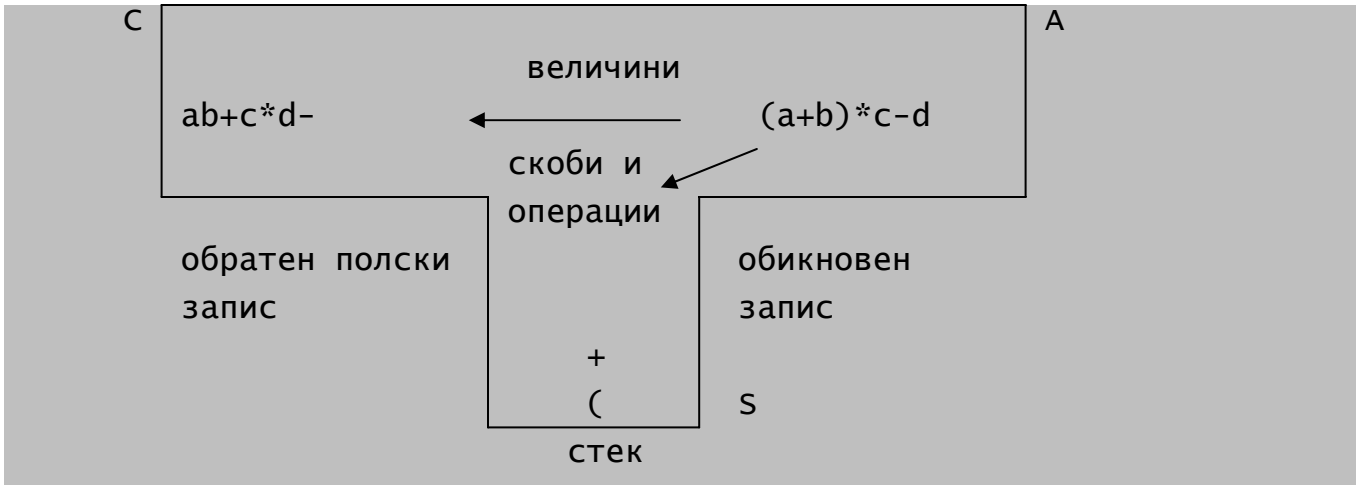
| *обикновен запис* | *обратен полски запис* |
|---|---|
| (a-b/c)*m | abc/-m* |

| | |
|---|---|
| a^n*b^m | an^bm^* |
| (a-b)*(a+b) | ab-ab+* |
| (a+b)*c-d*f+m^p | ab+c*df*-mp^+ |

Ако аритметичен израз е представен чрез обратен полски запис, пресмятането на стойността му се осъществява по следния начин. Сканира се изразът, представен чрез обратен полски запис отляво надясно до намиране на знак за операция. Пресмята се стойността на терма с аргументи – първите два, намиращи се непосредствено пред знака за операция. Знакът за операция и операндите се заменят с резултата от пресмятането, след което продължава търсенето на знак за операция. Сканирането продължава до достигане края на израза. За целта е удобно да се използва стек.

**Преобразуване на израз от обикновен в обратен полски запис**

Ще използваме стек. Фиг. 15.1 илюстрира преобразуването.




Фиг. 15.1 Преобразуване на аритметичен израз в обратен полски запис

Стекът е означен със S. Символите, участващи в обикновения запис се прехвърлят от частта A в частта C, като някои от тях временно престояват в стека S. В частта C се получава изразът, записан в обратен полски запис.

*Правилата за движение са следните:*

- Величините (цифри и променливи) се преместват директно от частта A в частта C.

- Скобата ‘(‘ се включва в стека S.
- Знаците за аритметични операции +, -, *, / и ^ се включват в стека S. Всяка операция има определен приоритет. Реализира се с цяло число, което се нарича тегло. Приемаме, че най-тежки са + и -, по-леки от тях са * и / и най-лека е операцията за степенуване. Ако при включване на знак за операция в стека S, под него има знак за операция с по-малко или с равно тегло, по-леката или с равно тегло операция се премества от S в частта C. Това се повтаря докато се достигне до по-тежка операция, до ‘)’ или до изпразването на стека.
- Скобата ‘)’ изключва от стека S всички знаци за операции до достигане до ‘(‘. Операциите се записват в частта C в реда на изключването им, а скобата ‘(‘ се унищожава от ‘)’.
- Ако всички символи от частта A са обработени, елементите на стека S, до изпразването му или до достигане до ‘(‘, се прехвърлят в областта C.

**Задача 131.** Да се напишат функции, реализиращи преобразуване на аритметичен израз от вида, описан по-горе, в обратен полски запис и също за намиране стойността на израз, представен в обратен полски запис.

Процедурата

```
void translate(char *s, char *ns);
```

превежда аритметичния израз, представен в обикновен запис чрез низа s, в обратен полски запис – низа ns. За да се избегне проверката за празен стек, в помощния стек още в началото включваме ‘(‘.

```
void translate(char *s, char *ns)
{stack<char> st;
 st.push('(');
 char x;
 int i = -1, j = -1, n = strlen(s);
 while (i < n)
 {i++;
```

```
 if (s[i] >= '0' && s[i] <= '9')
 {j++;
  ns[j] = s[i];
 }
 else
 if (s[i] == '(') st.push(s[i]);
 else
 if (s[i] == ')')
 {st.pop(x);
  while (x != '(')
  {j++;
   ns[j] = x;
   st.pop(x);
   }
  }
  else
  if (s[i] == '+' || s[i] == '-' ||
      s[i] == '*' || s[i] == '/' || s[i] == '^')
   {st.pop(x);
    while (x != '(' && t(x) <= t(s[i]))
    {j++;
     ns[j] = x;
     st.pop(x);
    }
    st.push(x);
    st.push(s[i]);
   }
  }
  st.pop(x);
  while (x != '(')
  {j++; ns[j] = x;
   st.pop(x);
  }
  j++;
  ns[j] = 0;
}
```

Функцията t намира теглото на операциите, използвани в аритметичния израз, определен по-горе и има вида:

```
int t(char c)
{int p;
 switch (c)
 {case '+': p = 2; break;
  case '-': p = 2; break;
  case '*': p = 1; break;
  case '/': p = 1; break;
  case '^': p = 0; break;
  default:  p = -1;
 }
 return p;
}
```

Функцията

```
int value(char *s);
```

намира стойността на израза, като използва обратния полски запис (представен чрез низа s), който му съответства.

```
int value(char *s)
{stack<int> st;   // генерира стек st от цели числа
 int x, y, z;
 unsigned int i = 0, n = strlen(s);
 while (i < n)
 {if (s[i] >= '0' && s[i] <= '9') st.push((int)s[i] - (int)'0');
  else
   if (s[i] == '+' || s[i] == '-' || s[i] == '*'
                   || s[i] == '/' || s[i] == '^')
   {st.pop(y);
    st.pop(x);
    switch (s[i])
    {case '+': z = x+y; break;
     case '-': z = x-y; break;
     case '*': z = x*y; break;
     case '/': z = x/y; break;
     case '^': z = (int)pow(x,y);
    }
```

```
      st.push(z);
      }
      i++;
      }
      st.pop(z);
      return z;
     }
```

Тогава намирането на стойността на аритметичен израз може да се реализира чрез изпълнението на следната главна функция:

```
     void main()
     {char s[200];
      cout << "s: ";
      cin >> s;
      char s1[200];
      translate(s, s1);
      cout << value(s1);
      cout << endl;
     }
```

**Задача 132.** В масив е записан без грешка булев израз от вида:

```
<булев_израз> ::= t | f | (~<булев_израз>)|
                  (<булев_израз>*<булев_израз>)|
                  (<булев_израз>+<булев_израз>)
```

където t означава истина, f – лъжа, а ~, * и + означават съответно логическо отрицание, конюнкция и дизюнкция. Да се напише програма, която намира стойността на правилно записан в масив булев израз.

Функцията BulFormula решава задачата. Тя използва два помощни стека s1 и s2. Ако сканираният символ е знак за операция, се включва в стека s1, а ако е t или f - в стека s2. Затварящата скоба изключва знак за операция от стека s1, анализира го и в зависимост от вида на операцията – унарна или бинарна изключва един или два елемента от s2. Изпълнява се операцията над съответните аргументи и резултатът се включва в стека s2. Останалите символи се пропускат.

```
// Program Zad132.cpp
#include <iostream.h>
```

```
#include <string.h>
#include “stack-link”
bool BulFormula(char* s)
{stack<char> s1, s2;
 char c, x, y;
 int i = -1, n = strlen(s);
 while (i < n)
 {i++;
  if (s[i] == '~' || s[i] == '*' || s[i] == '+') s1.push(s[i]);
  else
  if (s[i] == 't' || s[i] == 'f') s2.push(s[i]);
  else
  if (s[i] == ')')
  {s1.pop(c);
   switch(c)
   {case '~': s2.pop(x);
              if (x == 't') c = 'f'; else c = 't';
              s2.push(c); break;
    case '*': s2.pop(y); s2.pop(x);
              if (x == 't' && y == 't') c = 't'; else c = 'f';
              s2.push(c); break;
    case '+': s2.pop(y); s2.pop(x);
              if (x == 'f' && y == 'f') c = 'f'; else c = 't';
              s2.push(c); break;
    }
  }
 }
  s2.pop(c);
  return c == 't';
 }
void main()
{char s[200];
 cout << "s: "; cin >> s;
 cout << BulFormula(s) << endl;
 cout << endl;
}
```

## 15.2 Опашка

### 15.2.1 Дефиниране на опашка

**Логическо описание**

Опашката е крайна редица от елементи от един и същ тип. Операцията включване е допустима за елементите от единия край на редицата, който се нарича **край на опашката**, а операцията изключване на елемент – само за елементите от другия край на редицата, който се нарича **начало на опашката**. Възможен е достъп само до елемента, намиращ се в началото на опашката, при това достъпът е пряк.

При тази организация на логическите операции, първият включен елемент се изключва пръв. Затова опашката се определя още като структура от данни "*пръв влязъл – пръв излязъл*".

**Физическо представяне**

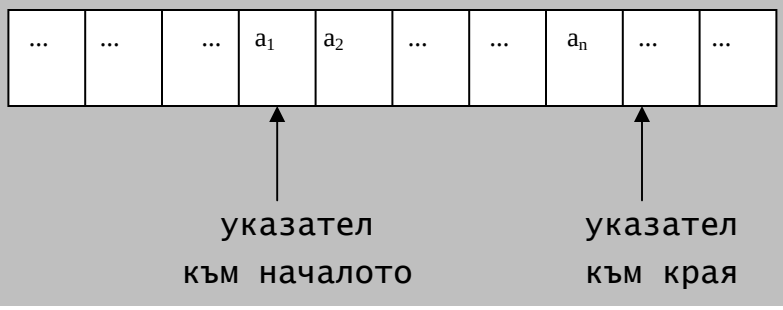
Широко се използват два основни начина за физическо представяне на опашка: *последователно* и *свързано*.

*Последователно представяне*

При това представяне се запазва блок от паметта, вътре в който опашката да расте и да се съкращава. Ако редицата от елементи от един и същ тип

$a_1, a_2, ..., a_n$

е опашка с начало $a_1$ и край $a_n$, последователното представяне има вида:

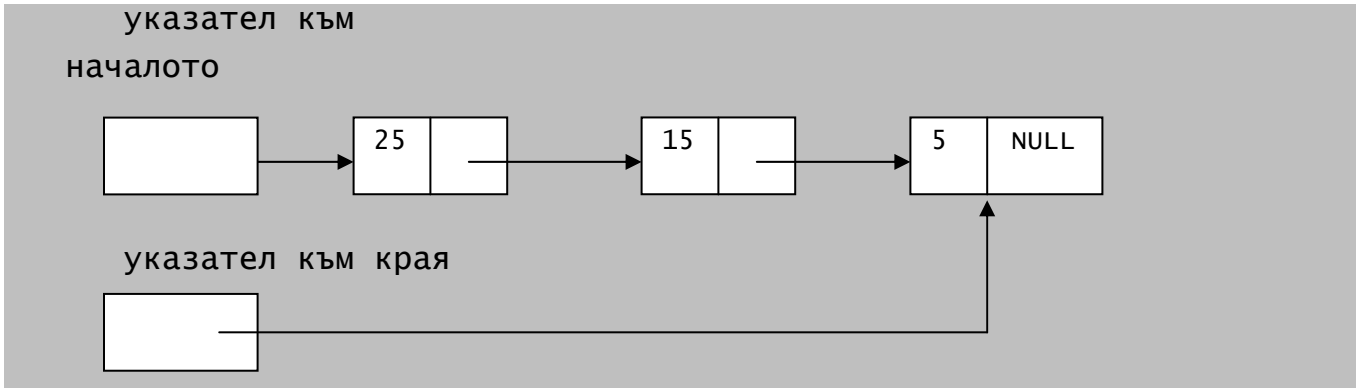



Фиг. 15.2 Последователно представяне на опашка

Включването на елемент в опашката се осъществява чрез поместването му в последователни адреси в неизползваната част веднага след края на опашката. При изчерпване на масива, ако има освободена памет в началото му, включването се извършва там.

*Свързано представяне*

Това представяне е аналогично на свързаното представяне на стек. За удобство, при реализирането на операцията включване, се въвежда указател и към края на опашката.




Фиг. 15.3 Свързано представяне на опашка с три елемента

## 15.2.2 Реализация на опашка

### 15.2.2.1 Реализация на последователното представяне

Ще изполваме динамичен масив с размер size. С front означаваме указателя към началото, с rear – указателя към края му, а с n – текущия брой на елементите на опашката. Ще дефинираме шаблон на клас, реализиращ последователно представяне на опашка.

```
const size = 100;
template <class T>
class queue
{public:
  queue();                    // конструктор
  ~queue();                   // деструктор
  queue(queue const &);       // конструктор за присвояване
```

```
  queue& operator=(queue const &);  // операторна функция за
                                    // присвояване
  void InsertElem(T const &);  // включване на елемент в опашка
  int DeleteElem(T &);        // изключване на елемент от опашка
  void print();              // извеждане на опашка
 private:
  int front, rear, n;
  T *a;
  void delqueue();               // изтриване на опашка
  void copy(queue const &);      // копиране на опашка
};
template <class T>
queue<T>::queue()
{a = new T[size];
 n = 0;
 front = 0;
 rear = 0;
}
template <class T>
queue<T>::~queue()
{delqueue();
}
template <class T>
queue<T>::queue(queue<T> const& r)
{copy(r);
}
template <class T>
queue<T>& queue<T>::operator=(queue<T> const& r)
{if (this != &r)
{delqueue();
 copy(r);
}
return *this;
}
template <class T>
void queue<T>::InsertElem(T const & x)
```

```
{if (n == size) cout << "Impossible! \n";
 else
 {a[rear] = x;
  n++; rear++;
  rear = rear % size;
 }
}
template <class T>
int queue<T>::DeleteElem(T &x)
{if (n > 0)
 {x = a[front]; n--; front++;
  front = front % size;
  return 1;
 }
 else return 0;
}
template <class T>
void queue<T>::delqueue()
{delete [] a;
}
template <class T>
void queue<T>::copy(queue<T> const &r)
{a = new T[size];
 for (int i = 0; i < size; i++)
    a[i] = r.a[i];
 n = r.n;
 front = r.front;
 rear = r.rear;
}
template <class T>
void queue<T>::print()
{T x;
 while (DeleteElem(x))
   cout << x << "  ";
 cout << endl;
}
```

Включваме този шаблон във файл с име queue.cpp. Следва програма, която използва дефинирания шаблон.

Забележете, файлът, съдържащ реализацията на шаблона на класа queue, е включен в програмата като заглавен файл, но е ограден в кавички, а не в “<” и “>” както е при системните заглавни файлове.

```
#include <iostream.h>
#include “queue.cpp”
void main()
{queue<int> q; // създава опашка от цели числа
 for (int i = 1; i <= 10; i++)   // запълва опашката
   q.InsertElem(i);              // с целите числа от 1 до 10
 q.print();                      // извежда опашката q
 queue<char> p;                  // създава опашка от символи
 for (char c = ’a’; i <= ’z’; i++)  // запълва опашката
    p.InsertElem(c);                // с малките латински букви
 p.print();                      // извежда опашката p
}
```

### 15.2.2.2 Реализация на свързаното представяне

Ще реализираме шаблон на клас queue, реализиращ свързаното представяне на опашка. Двойката

| inf | link |
|---|---|

ще реализираме чрез шаблона на структурата

```
template <class T>
struct elem
{T inf;
 elem* link;
};
```

а опашката – чрез шаблона на класа queue. Указателят към началото на опашката ще означим с front, а този към края – с rear. Отново се налага реализирането на голямата тройка (деструктор, конструктор за присвояване и операторна функция за присвояване). Реализицията на това представяне има вида:

```
template <class T>
struct elem
{T inf;
 elem* link;
};
template <class T>
class queue
{public:
  queue();
  ~queue();
  queue(queue const &);
  queue& operator=(queue const &);
  void InsertElem(T const&);
  int DeleteElem(T &);
  void print();
  bool empty() const;
 private:
  elem<T> *front, *rear;
  void delqueue();
  void copy(queue const&);
};
template <class T>   // конструктор
queue<T>::queue()
{front = NULL;
 rear = NULL;
}
template <class T>   // деструктор
queue<T>::~queue()
{delqueue();
}
template <class T>   // конструктор за присвояване
queue<T>::queue(queue const & r)
{copy(r);
}
template <class T>   // операторна функция за присвояване
```

```
queue<T>& queue<T>::operator=(queue const& r)
{if (this != &r)
 {delqueue();
  copy(r);
 }
 return *this;
}
template <class T>   // изтриване на опашка
void queue<T>::delqueue()
{T x;
 while (DeleteElem(x));
}
template <class T>      // копиране на опашка
void queue<T>::copy(queue const & r)
{rear = NULL;
 if (r.rear)
 {elem<T> *p = r.front;
  while (p)
  {InsertElem(p->inf);
   p = p->link;
  }
 }
}
template <class T>      // включване на елемент в опашка
void queue<T>::InsertElem(T const& x)
{elem<T> *p = new elem<T>;
 p->inf = x;
 p->link = NULL;
 if (rear) rear->link = p;
 else front = p;
 rear = p;
}
template <class T>   // изтриване на елемент от опашка
int queue<T>::DeleteElem(T & x)
{elem<T> *p;
 if (!rear) return 0;
```

```
 p = front;
 x = p->inf;
 if (p == rear)
 {rear = NULL;
  front = NULL;
 }
 else front = p->link;
 delete p;
 return 1;
}
template <class T>      // извеждане на опашка
void queue<T>::print()
{T x;
 while (DeleteElem(x))
   cout << x << " ";
 cout << endl;
}
template <class T>      // проверка за празна опашка
bool queue<T>::empty() const
{return rear == NULL;
}
```

Ще запишем този шаблон във файл с име queue-link.cpp, след което ще го включим в демонстрационна програма.

**Задача 133.** Да се напише програма, която създава и извежда опашка от опашки от цели числа.

Налага се предефинирането на член-функцията print() на шаблона на класа, тъй като основната член-функция print() извежда чрез използване на <<, с което не може да се изведе опашка.

```
// Program Zad133.cpp
#include <iostream.h>
#include “queue-link.cpp”
// дефиниране на клас IntQueue, реализиращ опашка от цели числа
typedef queue<int> IntQueue;
```

```
// дефиниране на клас QueueQueue, реализиращ опашка
// от опашки от цели числа
typedef queue<IntQueue> QueueQueue;
// предефиниране на шаблонната член-функция print()
// за извеждане на опашка от опашки.
void queue<IntQueue>::print() // специализация на член-функцията
{IntQueue x;                  // print() за извеждане на
 while (DeleteElem(x))        // опашка от опашки
   x.print();
 cout << endl;
}
void main()
{QueueQueue qq; // qq е обект, означаващ опашка от опашки
 for (int i = 1; i <= 5; i++)
  {IntQueue q;    // конструиране на опашка от цели числа
   for (int j = i; j <= 2*i; j++)
    q.InsertElem(j);
   qq.InsertElem(q); // включване на опашката q в опашката qq
  }
 qq.print();
}
```

## Задачи върху структурата от данни опашка

**Задача 134.** Да се напише програма, която само с едно преминаване през елементите на масив от числа (без използване на допълнителни масиви) извежда вурху екрана елементите на масива в следния ред: отначало всички числа, които са по-малки от a, след това всички числа в интервала [a, b] и накрая всички останали числа, запазвайки техния първоначален ред (a и b са дадени числа, a < b).

Програма Zad134.cpp решава задачата. Тя използва две опашки q1 и q2, в които записва числата от интервала [a, b] и тези – по-големи от b, съответно в реда на тяхното срещане в масива arr.

```
// Program Zad134.cpp
#include <iostream.h>
```

```
#include "queue-link.cpp"
typedef queue<int> IntQueue;
void read(int n, int *a)
{for (int i = 0; i < n; i++)
{cout << "a[" << i << "]= ";
 cin >> a[i];
}
}
void main()
{int arr[100];
 int n;
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 100);
 read(n, arr);
 cout << "a < b = ";
 int a, b;
 cin >> a >> b;
 IntQueue q1, q2;
 for (int i = 0; i < n; i++)
    if (arr[i] < a) cout << arr[i] << " ";
    else if (arr[i] <= b) q1.InsertElem(arr[i]);
    else q2.InsertElem(arr[i]);
  q1.print();
  q2.print();
}
```

**Задача 135.** Да се напише програма, която създава две опашки, елементите на които са структури, съдържащи име и възраст на човек. Сортира във възходящ ред по възраст елементите на опашките, след което ги слива и извежда получената опашка.

Програма Zad135.cpp решава задачата. За олесняване на сливането в края на всяка сортирана опашка добавяме фиктивен елемент, който се

нарича **сентинел**. Този трик често се използва при работа с линейни динамични структури.

```
// Program Zad135.cpp
#include <iostream.h>
#include "queue-link.cpp"
struct people
{char name[31];
 int age;
};
typedef queue<people> pqueue;
void queue<people>::print() // предефиниция на шаблонната функция
{people x;
 while (DeleteElem(x))
 cout << x.name << " " << x.age << endl;
}
// за опашката q се намират лицето с най-малка възраст (min) и
// опашката без лицето с най-малка възраст (newq)
// newq е инициализирана с празната опашка
void minqueue(pqueue q, people & min, pqueue &newq)
{people x;
 q.DeleteElem(min);
 while (q.DeleteElem(x))
  if (x.age < min.age)
  {newq.InsertElem(min);
   min = x;
  }
  else newq.InsertElem(x);
 }
// сортиране на опашката (q) във възходящ ред
// по възраст на лицата (nq)
void sortqueue(pqueue q, pqueue& nq)
{while (!q.empty())
 {people min;
  pqueue q1;
  minqueue(q, min, q1); // q1 е инициализирана с празната опашка
```

```
  nq.InsertElem(min);
  q = q1;
 }
}
// сливане на сортираните опашки p и q
// връща резултата от сливането
pqueue merge(pqueue p, pqueue q)
{people x = {"xxx", -1}, y = {"yyy", -1}; // фиктивни елементи
 p.InsertElem(x); // включване на фиктивния елемент x в опашката p
 q.InsertElem(y); // включване на фиктивния елемент y в опашката q
 pqueue r;
 p.DeleteElem(x);
 q.DeleteElem(y);
 while (!p.empty() && !q.empty())
   if (x.age <= y.age)
   {r.InsertElem(x);
    p.DeleteElem(x);
   }
   else
   {r.InsertElem(y);
    q.DeleteElem(y);
   }
 if (!p.empty()) // q е празна
    do
      r.InsertElem(x);
    while (p.DeleteElem(x) && x.age != -1);
 else // p е празна
    do
      r.InsertElem(y);
    while (q.DeleteElem(y) && y.age != -1);
 return r;
}
void main()
{people s;
 pqueue q1;
 int i, n;
```

```
 cout << "First Queue:\n n= ";
 cin >> n;
 for (i = 0; i < n; i++)
 {cout << "Name: ";
  cin>> s.name;
  cout << "Age: ";
  cin >> s.age;
  q1.InsertElem(s);
 }
 pqueue q2;
 cout << "Second Queue \n n= ";
 cin >> n;
 for (i = 0; i < n; i++)
 {cout << "Name: ";
  cin>> s.name;
  cout << "Age: ";
  cin >> s.age;
  q2.InsertElem(s);
 }
 pqueue p, q, r;
 sortqueue(q1, p);
 sortqueue(q2, q);
 r = merge(p, q);
 r.print();
}
```

**Задача 136.** Да се напише програма, която създава опашка от опашки, елементите на които са структури, съдържащи име и възраст на човек. Да се сортират по възраст компонентите на опашката, след което да се слеят.

Програма Zad136.cpp решава задачата. Някои функции в нея са пропуснати, тъй като са същите като в задача 135.

```
// Program Zad136.cpp
#include <iostream.h>
#include "queue-link.cpp"
```

```
struct people
{char name[31];
 int age;
};
typedef queue<people> pqueue;
void queue<people>::print()
{people x;
 while (DeleteElem(x))
    cout << x.name << " "
         << x.age << endl;
}
void queue<pqueue>::print()
{pqueue x;
 while (DeleteElem(x))
   {x.print();
    cout << endl;
   }
}
void minqueue(pqueue q, people & min, pqueue &newq)
...
void sortqueue(pqueue q, pqueue& nq)
...
pqueue merge(pqueue p, pqueue q)
...
void main()
{queue<pqueue> q2, q3;
 int m;
 cout << “m= “; cin >> m;
 for (int j = 1; j <= m; j++)
 {int i, n;
  pqueue q1;
  cout << j << " Queue:\n n= ";
  cin >> n;
  for (i = 0; i < n; i++)
  {people s;
   cout << "Name: ";
```

```
      cin >> s.name;
      cout << "Age: ";
      cin >> s.age;
      q1.InsertElem(s);
     }
    q2.InsertElem(q1);
    }
    pqueue r;
    while (q2.DeleteElem(r))
    {pqueue t;
     sortqueue(r, t);
     q3.InsertElem(t);
    }
    pqueue p1, p2;
    q3.DeleteElem(p1);
    while (q3.DeleteElem(p2))
       p1 = merge(p1, p2);
    p1.print();
   }
```

## 15.3 Свързан списък

При стековете и опашките единственият начин за извличане на данни от структурата се осъществява чрез отсраняване на елементи. Често се налага да се използват всички данни от редица от елементи без да се отстраняват елементите от редицата. За целта се използва структурата от данни свързан списък.

### 15.3.1 Дефиниране на свързан списък

**Логическо описание**

Свързаният списък е крайна редица от елементи от един и същ тип. Операциите включване и изключване са допустими в произволно място на редицата. Възможен е пряк достъп до елемента в единия край на

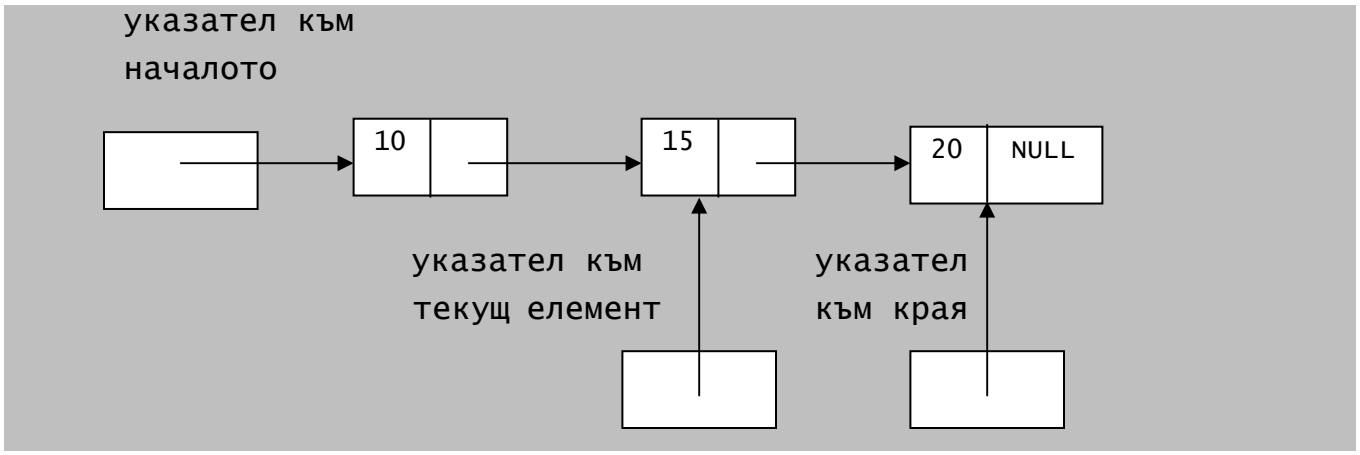
редицата, наречен **начало на списъка**, и последователен до всеки от останалите елементи.

**Физическо представяне**

Има два основни начина за представяне на свързания списък в паметта на компютъра: *свързано представяне с една* и *свързано представяне с две връзки*. Възможно е и *последователно представяне* (чрез масив от структури), което не се използва напоследък заради добре развитите средства за динамично разпределение на паметта.

*Свързано представяне с една връзка*

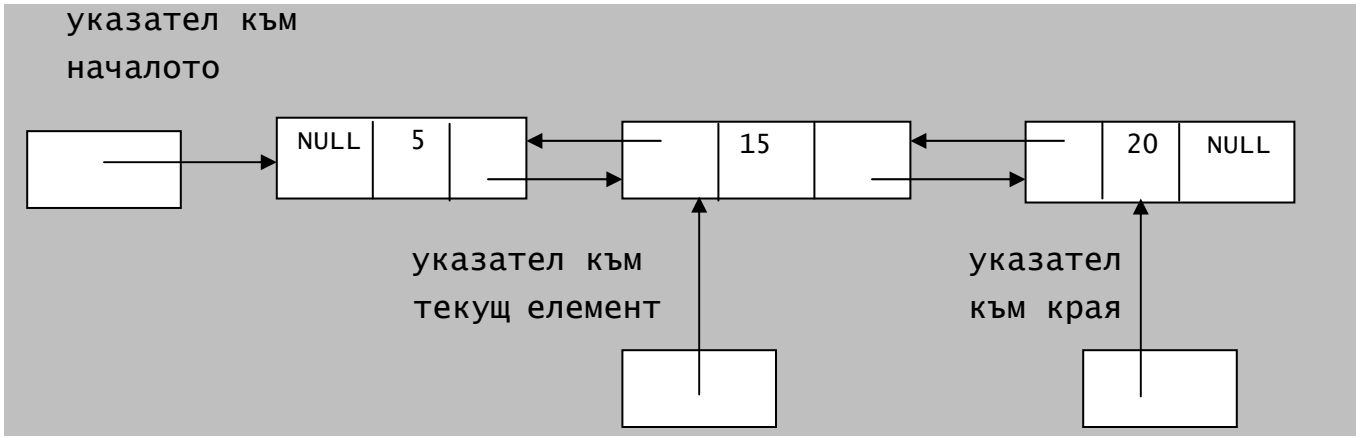
Използва се представяне, аналогично на свързаното представяне на стек и опашка (Фиг. 15.4). За удобство, при реализирането на операциите включване, изключване и обхождане, се въвеждат и указатели към края и към текущ елемент на списъка.




Фиг. 15.4 Представяне на свързан списък с една връзка

*Свързано представяне с две връзки*

За удобство, при реализирането на операциите включване, изключване и обхождане, се въвеждат тройни кутии, с едно информационно и две свързващи полета, съдържащи текущия елемент и адресите на предшестващия и следващия го елементи на свързания списък (Фиг. 15. 5).




Фиг. 15. 5 Представяне на свързан списък с две връзки

## 15.3.2 Реализация на представяне на свързан списък с една връзка

Следният шаблон на клас реализира това представяне на свързан списък.

```
template <class T>
struct elem
{T inf;
 elem *link;
};
template <class T>
class LList
{public:
  LList();
  ~LList();
  LList(LList const&);
  LList& operator=(LList const &);
  void print();
  void IterStart(elem<T>* = NULL);
  elem<T>* Iter();
  void ToEnd(T const &);
  void InsertAfter(elem<T> *, T const &);
  void InsertBefore(elem<T> *, T const &);
  int DeleteAfter(elem<T> *, T &);
```

```
  int DeleteBefore(elem<T> *, T &);
  void DeleteElem(elem<T> *, T &);
  int len();
  void concat(LList const&);
  void reverse();
 private:
  elem<T> *Start,       // указател към началото
          *End,         // указател към края
          *Current;     // указател към текущ елемент
  void DeleteList();
  void CopyList(LList<T> const &);
};
```

Конструктора и член-функциите на голямата тройка няма да коментираме. Реализират аналогични идеи на тези при стека и опашката.

```
template <class T>   // конструктор
LList<T>::LList()
{Start = NULL;
 End = NULL;
}
template <class T>   // деструктор
LList<T>::~LList()
{DeleteList();
}
template <class T>   // конструктор за присвояване
LList<T>::LList(LList<T> const& r)
{CopyList(r);
}
template <class T>   //операторна функция за присвояване
LList<T>& LList<T>::operator=(LList<T> const & r)
{if (this != &r)
 {DeleteList();
  CopyList(r);
 }
 return *this;
}
```

Следващите две функции са помощни и реализират изтриване на свързан списък и копиране на свързан списък на друго място в паметта. Изпозват се за реализиране на голямата тройка и затова са капсолирани в private-частта на класа. Макар, че могат да се реализират с помощта на член-функции на шаблона, по-добра е реализацията им без тях – чрез обхождане на елементите им.

*Изтриване на списък*

```
template <class T>
void LList<T>::DeleteList()
{elem<T> *p;
 while (Start)
 {p = Start;
  Start = Start->link;
  delete p;
 }
 End = NULL;
}
```

Следващите реализации на тази член-функция на шаблона използват член-функциите DeleteAfter, Delete Before и DeleteElem.

```
template <class T>
void LList<T>::DeleteList()
{if (End)
 {T x;
  while (DeleteAfter(Start, x));
  DeleteElem(Start, x);
 }
}
```

и

```
template <class T>
void LList<T>::DeleteList()
{if (Start)
 {T x;
  while (DeleteBefore(End, x));
  DeleteElem(Start, x);
 }
}
```

*Копиране на списък*

```
template <class T>
void LList<T>::CopyList(LList<T> const & r)
{elem<T> *p = r.Start, *q;
 if (r.Start)
 {Start = new elem<T>;
  End = Start;
  while (p)
  {End->inf = p->inf;
   End->link = NULL;
   p = p->link;
   if (p)
   {q = End;
    End = new elem<T>;
    q->link = End;
   }
  }
 }
}
```

Друга, по-лесна реализация на тази член-функция на шаблона е:

```
template <class T>
void LList<T>::CopyList(LList<T> const & r)
{Start = End = NULL;
 if (r.Start)
 {elem<T> *p = r.Start;
  while (p)
  {ToEnd(p->inf);
   p = p->link;
  }
 }
}
```

и я избираме за реализация на копирането.

*Извеждане на елементите на списък*

Извеждането на елементите на свързан списък се реализира чрез последователното им обхождане, без разрушаването на списъка.

```
template <class T>
```

```
void LList<T>::print()
{elem<T> *p = Start;
 while (p)
 {cout << p->inf << " ";
  p = p->link;
 }
 cout << endl;
}
```

Член-функциите IterStart и Iter реализират операциите с т. нар. **итератори**.

Итераторът е абстракция на означението указател към елемент на редица или по-точно може да се смята за указател към елемент на контейнер (стекът, опашката, свързаният списък са контейнери). Всеки конкретен итератор е обект (в широкия смисъл на думата) от някакъв тип. Разнообразието на типове води до разнообразие на итераторите. В някои случаи итераторите са почти обикновени указатели към обекти, в други – са указател, снабден с индекс и т.н. В случая на свързан списък итераторът е указател към двойна или тройна кутия. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:

++ - приложена към итератор, намира итератор,
      който сочи към следващия елемент;

-- - приложена към итератор, намира итератор,
      който сочи към предшестващия елемент;

* - намира елемента, към който сочи итераторът.

В шаблона на класа LList итераторът е означен с Current. Операцията ++ е реализирана чрез член-функцията Iter. За получаване на указател към началото на свързан списък, е използвана процедурата IterStart. Тя е с един подразбиращ се параметър. Подразбиращата се стойност е NULL. Обръщението IterStart() установява указателя Current в началото на текущия списък. В случай, че е указан ненулев параметър, Current се установява в него. Операцията * не е реализирана, тъй като Current сочи елемент, който е реализиран като структура, но обръщението Iter()->inf я реализира.

*Установяване на итератора в началото*

```
template <class T>
void LList<T>::IterStart(elem<T> *p)
{if (p) Current = p;
 else Current = Start;
}
```

Преместването на указателя Current в следващата позиция се осъществява чрез член-функцията Iter.

*Установяване на итератора в следващата позиция*

```
template <class T>
elem<T>* LList<T>::Iter()
{elem<T> *p = Current;
 if (Current) Current = Current->link;
 return p;
}
```

Като използваме член-функциите, реализиращи основните операции за работа с итератора, член-функцията print() на шаблона LList може да се реализира и по следния начин:

```
template <class T>
void LList<T>::print()
{IterStart();
 while (Iter())
 {cout << Iter()->inf << " ";
 }
 cout << endl;
}
```

Включването на елемент в свързан списък реализираме чрез следните три член-функции:

*Включване на елемент в края на списъка*

```
template <class T>
void LList<T>::ToEnd(T const & x)
{Current = End;
 End = new elem<T>;
 End->inf = x;
```

```
 End->link = NULL;
 if (Current) Current->link = End;
 else Start = End;
}
```

*Включване на елемент след указан елемент*

```
template <class T>
void LList<T>::InsertAfter(elem<T> *p, T const & x)
{elem<T> *q = new elem<T>;
 q->inf = x;
 q->link = p->link;
 if (p == End) End = q;
 p->link = q;
}
```

*Включване на елемент пред указан елемен*

```
template <class T>
void LList<T>::InsertBefore(elem<T> * p, T const& x)
{elem<T> *q = new elem<T>;
 *q = *p;
 if (p == End) End = q;
 p->inf = x;
 p->link = q;
}
```

Изтриването на елемент от свързан списък реализираме чрез следните член-функции:

*Изтриване на елемент след указан елемент*

```
template <class T>
int LList<T>::DeleteAfter(elem<T> *p, T &x)
{if (p->link)
{elem<T> *q = p->link;
 x = q->inf;
 p->link = q->link;
 if (q == End) End = p;
 delete q;
 return 1;
```

```
 }
 else return 0;
}
```

*Изтриване на указан елемент*

```
template <class T>
void LList<T>::DeleteElem(elem<T> *p, T &x)
{if (p == Start)
 {x = p->inf;
  if (Start == End)
  {Start = End = NULL;
  }
  else Start = Start->link;
  delete p;
 }
 else
 {elem<T> *q = Start;
  while (q->link != p) q = q->link;
  DeleteAfter(q, x);
 }
}
```

*Изтриване на елемент пред указан елемент*

```
template <class T>
int LList<T>::DeleteBefore(elem<T> *p, T &x)
{if (p != Start)
 {elem<T> *q=Start;
  while (q->link != p) q = q->link;
  DeleteElem(q, x);
  return 1;
 }
 else return 0;
}
```

*Дължина на списък*

Член функцията len намира броя на елементите на свързан списък.

```
template <class T>
int LList<T>::len()
```

```
{int n = 0;
 IterStart();
 elem<T> *p = Iter();
 while (p)
 {n++;
  p = Iter();
 }
 return n;
}
```

или

```
template <class T>
int LList<T>::len()
{int n = 0;
 elem<T> *p = Start;
 while (p)
 {n++;
  p = p->link;
 }
 return n;
}
```

*Конкатенация на списъци*

Следващата член-функция реализира конкатенация на неявния свързан списък с указания като формален параметър. В резултат в края на неявния списък са включени елементите на указания списък. Така неявният списък е разрушен (носи резултата).

```
template <class T>
void LList<T>::concat(LList<T> const &L)
{elem<T> *p = L.Start;
 while (p)
 {ToEnd(p->inf);
  p = p->link;
 }
}
```

Често пъти вместо тази функция се използва следната:

```
template <class T>
void LList<T>::concat(LList const& L)
```

```
{End->link = L.Start;
}
```

Тя е неправилна, тъй като една и съща редица от елементи е достъпна чрез два обекта на класа LList, т.е. имат поделена част.

*Обръщане на елементите на списък*

Следващата член-функция на класа LList обръща елементите на неявния параметър като ползва помощен списък, с който работи като със стек.

```
template <class T>
void LList<T>::reverse()
{LList<T> l;
 IterStart();
 elem<T> *p = Iter();
 if (p)
 {l.ToEnd(p->inf);
  p = p->link;
  while (p)
  {l.InsertBefore(l.Start, p->inf);
   p = p->link;
  }
 }
 *this = l;
}
```

Реализацията на reverse, дадена по-долу, обръща елементите на непразен списък, зададен чрез неявния параметър без да прави негово копие в паметта. Тя поправя връзките в него така, че първите елементи да станат последни и обратно.

```
template <class T>
void LList<T>::reverse()
{elem<T> *p, *q, *temp;
 p = Start;
 if (p)
 {q = NULL;
  temp = Start;
  Start = End;
  End = temp;
```

```
  while (p != Start)
  {temp = p->link;
   p->link = q;
   q = p;
   p = temp;
  }
  p->link = q;
 }
}
```

### 15.3.3 Приложения на свързните списъци, представени с една връзка

Шаблонът на класа LList записваме във файла L-List.cpp.

**Общи приложения**

**Задача 137.** Да се напише програма, която създава свързан списък с една връзка по различни начини, обръща елементите на списъка и ги извежда.

```
// Program Zad137.cpp
#include <iostream.h>
#include "L-List.cpp"
void main()
{LList<int> l1, l2;
 // създава списъка l1 с елементи 1, 2, 3, 4
 l1.ToEnd(1); l1.ToEnd(2);
 l1.ToEnd(3); l1.ToEnd(4);
 // създава списъка l2 с елементи 5, 6
 l2.ToEnd(5); l2.ToEnd(6);
 // създава списъка l3 чрез конструктора за присвояване
 LList<int> l3 = l1;
 // извежда списъците
 cout << "l1= "; l1.print();
 cout << "l2= "; l2.print();
 cout << "l3= "; l3.print();
```

```
 // обръща елементите на списъка l1
 l1.reverse();
 cout << "L1-rev: "; l1.print();
}
```

**Задача 138.** Да се напише програма, която като използва класа LList моделира стек.

```
// Program Zad138.cpp
#include <iostream.h>
#include "L-List.cpp"
void main()
{// създаване на стека от цели числа 9,8,..., 0 с връх 9
 LList<int> l1;
 l1.ToEnd(0);
 l1.IterStart();
 elem<int>* p = l1.Iter();
 for (int i = 1; i <= 9; i++)
   l1.InsertBefore(p, i);
 // изключване на елементи от стека
 l1.IterStart();
 p = l1.Iter();
 int x;
 l1.DeleteElem(p, x);
 cout << x << " ";
 l1.IterStart();
 p = l1.Iter();
 l1.DeleteElem(p, x);
 cout << x << " ";
}
```

**Задача 139.** Да се напише програма, която като използва класа LList моделира опашка.

```
// Program Zad139.cpp
#include <iostream.h>
#include "L-List.cpp"
```

```
void main()
{LList<int> l1;
 l1.ToEnd(1);
 l1.IterStart();
 elem<int>* p = l1.Iter();
 for (int i = 2; i <= 10; i++)
   l1.ToEnd(i);
 l1.print();
 l1.IterStart();
 p = l1.Iter();
 int x;
 l1.DeleteElem(p, x);
 cout << "x= " << x << endl;
 l1.IterStart();
 p = l1.Iter();
 l1.DeleteElem(p, x);
 cout << "x= " << x << endl;
 l1.print();
}
```

**Задача 140.** Да се дефинира шаблон на функция, която реализира увеличаване на елементите на свързан списък от тип T с елемента a от тип T (T е числов тип).

```
template <class T>
LList<T> increase(LList<T> L, T a)
{elem<T> *p;
 L.IterStart();
 p = L.Iter();
 while (p)
 {p->inf = p->inf + a;
  p = L.Iter();
 }
 return L;
}
```

**Задача 141.** Даден е списък от цели числа. Да се напише функция, която:

а) изтрива първото срещане на дадено цяло число;

б) изтрива всяко срещане на дадено цяло число.

Ще направим следната специализация на класа LList.

```
typedef LList<int> IntList;
а)
void deletefirst(int a, IntList& l)
{int x;
 l.IterStart();
 elem<int> *p = l.Iter();
 while (p && p->inf != a) p = l.Iter();
 if (p->inf == a) l.DeleteElem(p, x);
}
```

или

```
void deletefirst(int a, IntList& l)
{l.IterStart();
 elem<int> *p = l.Iter();
 while (p)
 if(p->inf == a)
 {int x;
  l.DeleteElem(p, x);
  p = NULL;
 }
 else p = l.Iter();
}
б)
void deleteall(int a, IntList& l)
{int x;
 l.IterStart();
 elem<int> *p = l.Iter();
 while (p)
 {if (p->inf == a) l.DeleteElem(p, x);
  p = l.Iter();
 }
```

```
   }
```
или
```
   void deleteall(int a, IntList& l)
   {while (member(a, l))
      deletefirst(a, l);
   }
```
където member(a, l) е функция, проверяваща дали a принадлежи на списъка l. Една нейна дефиниция е дадена в следващата задача.

**Някои рекурсивни функции за работа със списъци с една връзка**

**Задача 142.** Даден е списък от цели числа. Да се напише рекурсивна функция, която:

а) проверява дали дадено цяло число се съдържа в списъка;
б) намира максималния елемент на списъка;
в) изтрива от списъка първото срещане на дадено цяло число;
г) изтрива от списъка всяко срещане на дадено цяло число;
д) извежда в обратен ред елементите на списъка.

Ще използваме следната специализация на класа LList.
```
   typedef LList<int> IntList;
```
а)
```
   bool member(int x, IntList l)
   {l.IterStart();
    elem<int> *p = l.Iter();
    if (!p) return false;
    int y;
    l.DeleteElem(p, y);
    return x == y || member(x, l);
   }
```
б)
```
   int maxelem(IntList l)
   {int x;
    l.IterStart();
    elem<int> *p = l.Iter();
    if (!p->link) return p->inf;
```

```
    l.DeleteElem(p, x);
    int y = maxelem(l);
    if (x >= y) return x;
    else return y;
   }
```

в)

```
   void deletefirst(int a, IntList &l)
   {l.IterStart();
    elem<int>*p = l.Iter();
    if (p)
    {int x;
     l.DeleteElem(p, x);
     if (x == a) return;
     else
     {deletefirst(a, l);
      l.IterStart(); p = l.Iter();
      if (p) l.InsertBefore(p, x);
      else l.ToEnd(x);
     }
    }
   }
```

г)

```
   void deleteall(int a, IntList &l)
   {l.IterStart();
    elem<int>*p = l.Iter();
    if (p)
    {int x;
     l.DeleteElem(p, x);
     if (x == a) deleteall(a, l);
     else
     {deleteall(a, l);
      l.IterStart();
      p = l.Iter();
      if (p) l.InsertBefore(p, x);
      else l.ToEnd(x);
     }
```

```
    }
   }
д) void print_reverse(IntList l)
   {int x;
    l.IterStart();
    elem<int> *p = l.Iter();
    if (p)
    {l.DeleteElem(p, x);
     print_reverse(l);
     cout << x << " ";
    }
   }
```

**Сортиране и сливане на списъци**

*Пряка селекция*

**Задача 143.** Да се дефинира шаблон на функция, който реализира метода на пряката селекция за сотриране на списъци с елементи от тип T, допускащи сравнение.

```
template <class T>
void sortlist(LList<T> &l)
{elem<T>* mp, *p;
 l.IterStart(); p = l.Iter();
 while (p->link)
 {T min = p->inf;
  mp = p;
  elem<T> *q = p->link;
   while (q)
   {if (q->inf <= min)
   {mp = q;
    min = q->inf;
   }
    q = q->link;
   }
   min = mp->inf;
```

```
      mp->inf = p->inf;
      p->inf = min;
      p = p->link;
     }
   }
```

*Сливане на сортирани свързани списъци*

**Задача 144.** Да се дефинира шаблон на функция mergelists, която реализира сливане на списъците l1 и l2 от тип T и връща резултата от сливането.

```
template <class T>
LList<T> mergelists(LList<T> l1, LList<T> l2)
{LList<T> l;
 l1.IterStart();
 l2.IterStart();
 elem<T> *p = l1.Iter(),
         *q = l2.Iter();
 while (p && q)
 if (p->inf <= q->inf)
 {l.ToEnd(p->inf);
  p = l1.Iter();
 }
 else
 {l.ToEnd(q->inf);
  q = l2.Iter();
 }
 if (q)
 while (q)
 {l.ToEnd(q->inf);
  q = l2.Iter();
 }
 else while(p)
 {l.ToEnd(p->inf);
  p = l1.Iter();
```

```
 }
 return l;
}
```

**Задача 145.** Да се напише програма, която създава списък от списъци от имена на хора, сортира компонентите на списъка, след което ги слива и извежда получения списък.

```
Програма Zad145.cpp решава задачата.
Program Zad145.cpp
#include <iostream.h>
#include <string.h>
#include "L-List.cpp"
typedef char str[31];
// дефиниране на клас strlist,
// реализиращ списък от имена на хора
typedef LList<str> strlist;
// дефиниране на клас list_of_lists,
// реализиращ списък от списъци от имена на хора
typedef LList<strlist> list_of_lists;
// специализация на член-функцията print() на шаблона
// за извеждане на списък от списъци от имена на хора
void LList<strlist>::print()
{elem<strlist> *p = Start;
 while (p)
 {p->inf.print();
  p = p->link;
 }
}
// специализация на член-функцията за включване на
// елемент в края на свързан списък
void LList<str>::ToEnd(str const & x)
{Current = End;
 End = new elem<str>;
 strcpy(End->inf, x);
 End->link = NULL;
```

```
 if (Current) Current->link = End;
 else Start = End;
}
// предефиниране на нова sortlist
// за сортиране на списък от хора
void sortlist(strlist &l)
{elem<str>* mp, *p;
 l.IterStart(); p = l.Iter();
 while (p->link)
 {str min;
  mp = p;
  strcpy(min, p->inf);
  elem<str> *q = p->link;
  while (q)
  {if (strcmp(q->inf, min) <= 0)
  {mp = q;
   strcpy(min, q->inf);
  }
   q = q->link;
  }
   strcpy(min, mp->inf);
   strcpy(mp->inf, p->inf);
   strcpy(p->inf, min);
   p = p->link;
  }
}
// предефиниране на mergelists за сливане на
// сортирани списъци от имена на хора
strlist mergelists(strlist l1, strlist l2)
{strlist l;
 l1.IterStart();
 l2.IterStart();
 elem<str> *p = l1.Iter(),
           *q = l2.Iter();
 while (p && q)
    if (strcmp(p->inf, q->inf) <= 0)
```

```
    {l.ToEnd(p->inf);
     p = l1.Iter();
    }
    else
    {l.ToEnd(q->inf);
     q = l2.Iter();
    }
 if (q)
    while (q)
    {l.ToEnd(q->inf);
     q = l2.Iter();
    }
else
   while (p)
   {l.ToEnd(p->inf);
    p = l1.Iter();
   }
 return l;
}
void main()
{// създаване на списък от списъци от хора
 list_of_lists ll;
 cout << "брой на списъците в списъка от списъци: ";
 int n;
 cin >> n;
 for (int k = 1; k <= n; k++)
 {strlist l;
  str s;
  cout << "брой на хората в списъка: ";
  int p;
  cin >> p;
  for (int i = 1; i <= p; i++)
  {cout << "s= ";
   cin >> s;
   l.ToEnd(s);
  }
```

```
     ll.ToEnd(l);
    }
    ll.print();
    // обхождане на елементите на ll и
    // сортиране на всеки от съставящите го списъци
    ll.IterStart();
    elem<strlist> *p = ll.Iter();
    while (p)
    {sortlist(p->inf);
     p = ll.Iter();
    }
    ll.print();
    // сливане на списъците, изграждащи ll
    ll.IterStart();
    p = ll.Iter();
    strlist l = p->inf;
    p = ll.Iter();
    while (p)
    {l = mergelists(l, p->inf);
     p = ll.Iter();
    }
    l.print();
   }
```

*Сортиране чрез сливане*

Този вид сортиране се осъществява по следния начин: списъкът се разделя на две половини. Всяка половинка се сортира и накрая сортираните половинки се сливат.

**Задача 146.** Да се напише шаблон на функция, който реализира метода сортиране чрез сливане.

Шаблонът mergesort реализира този начин за сортиране.

```
template <class T>
void mergesort(LList<T> &l)
{LList<T> l1, l2;
```

```
 l.IterStart();
 elem<T> *p = l.Iter();
 if (!p || p->link == NULL) return;
 while (p)
 {l1.ToEnd(p->inf);
  p = p->link;
  if (p)
  {l2.ToEnd(p->inf);
   p = p->link;
  }
 }
 mergesort(l1);
 mergesort(l2);
 l = mergelists(l1, l2);
}
```

**Проверка на свойства на списъци**

**Задача 147.** Да се напише програма, която създава списък от имена на хора и проверява дали елементите му са подредени лексикографски.

Програма Zad147.cpp решава задачата.

```
// Program Zad147.cpp
#include <iostream.h>
#include <string.h>
#include "L-List.cpp"
typedef char str[31];
typedef LList<str> strlist;
void LList<strlist>::print()
{elem<strlist> *p = Start;
 while (p)
 {p->inf.print();
  p = p->link;
 }
}
void LList<str>::ToEnd(str const & x)
```

```
{Current = End;
 End = new elem<str>;
 strcpy(End->inf, x);
 End->link = NULL;
 if (Current) Current->link = End;
 else Start = End;
}
bool monotone(strlist l)
{l.IterStart();
 elem<str>  *p = l.Iter(),
            *q,
            *r;
 if (!p->link) return true;
 q = p->link;
 r = q->link;
 while (strcmp(p->inf, q->inf) <= 0 && r)
 {p = q;
  q = r;
  r = r->link;
 }
 return strcmp(p->inf, q->inf) <= 0;
}
void main()
{strlist l;
 str s;
 cout << "broj= ";
 int p;
 cin >> p;
 for (int i = 1; i <= p; i++)
 {cout << "s= ";
  cin >> s;
  l.ToEnd(s);
 }
 l.print();
 cout << monotone(l) << endl;
}
```

**Задача 148.** Да се напише програма, която създава списък от цели числа и проверява дали елементите му са:

а) монотонно намаляващи;

б) различни.

Програма Zad148.cpp решава задачата.

```
// Program Zad148.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<int> IntList;
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if (!p) return false;
 int y;
 l.DeleteElem(p, y);
 return x == y || member(x, l);
}
// a)
bool monot(IntList l)
{l.IterStart();
 elem<int>*p = l.Iter();
 if (!p->link) return true;
 int x;
 l.DeleteElem(p, x);
 IntList l1 = l;
 l.IterStart();
 p = l.Iter();
 int y;
 l.DeleteElem(p, y);
 return x >= y && monot(l1);
}
// б)
bool diffr(IntList l)
{l.IterStart();
```

```
 elem<int> *p = l.Iter();
 if (!p->link) return true;
 int y;
 l.DeleteElem(p, y);
 return !member(y, l) && diffr(l);
}
void main()
{IntList l;
 int s;
 cout << "broj= ";
 int p;
 cin >> p;
 for (int i = 1; i <= p; i++)
 {cout << "s= ";
  cin >> s;
  l.ToEnd(s);
  }
cout << diffr(l) << endl;
}
```

**Конструиране на списък от елементи на други списъци**

В следващата задача се реализират операциите обединение, сечение и разлика на множества, представени чрез списъци.

**Задача 149.** Дадени са два списъка от цели числа p и q. Да се дефинира функция, която намира списък от цели числа, съдържащ елементите, които:

а) принадлежат на поне един от списъците p или q;

б) принадлежат едновременно и на p, и на q;

в) принадлежат на p, но не принадлежат на q;

г) принадлежат на един от списъците, но не принадлежат на другия.

Някои от тези функции са реализирани в програма Zad150.cpp.

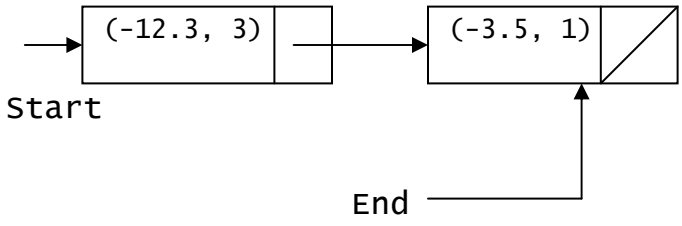
```
// Program Zad149.cpp
#include <iostream.h>
```

```
#include "L-List.cpp"
typedef LList<int> IntList;
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if (!p) return false;
 int y;
 l.DeleteElem(p, y);
 return x == y || member(x, l);
}
// a)
void unilists(IntList p, IntList q, IntList &r)
{p.IterStart();
 q.IterStart();
 elem<int> *pp = p.Iter(),
           *qq = q.Iter();
 while (pp)
 {if (!member(pp->inf, r)) r.ToEnd(pp->inf);
  pp = pp->link;
 }
 while (qq)
 {if (!member(qq->inf, r)) r.ToEnd(qq->inf);
  qq = qq->link;
 }
}
// б)
void intersections(IntList p, IntList q, IntList &r)
{p.IterStart();
 q.IterStart();
 elem<int> *pp = p.Iter(),
           *qq = q.Iter();
 while (pp)
 {if (!member(pp->inf, r) && member(pp->inf, q))
   r.ToEnd(pp->inf);
  pp = pp->link;
 }
```

```
}
void read(IntList & l)
{int s;
 cout << "number= ";
 int p; cin >> p;
 for (int i = 1; i <= p; i++)
 {cout << "elem= ";
  cin >> s;
  l.ToEnd(s);
  }
}
void main()
{IntList l, l1, l2, l3;
 read(l1);
 cout << "L1: "; l1.print();
 read(l2);
 cout << "L2: "; l2.print();
 unilists(l1, l2, l);
 l.print();
 intersections(l1, l2, l3);
 l3.print();
}
```

**Използване на свързан списък за реализиране и работа с полиноми**

**Задача 150.** Полимомът $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ може да се представи чрез списък елементите, на който са структури с две полета: coef, означаващо коефициента и pow, означаващо степента на съответния едночлен. При това, ако коефициентът $a_k$ е 0, съответният едночлен не се включва в списъка. Например, полиномът $-12.3x^3 - 3.5x$ се представя по следния начин:



Да се напишат следните функции за работа с полиноми:

а) въвеждане на полином;
б) извеждане на полином;
в) намиране на стойността на полином за дадено x;
г) намиране на производната на полином;
д) намиране на интеграла на полином;
е) намиране на сумата на два полинома;
ж) намиране на произведението на полином с едночлен;
з) намиране на произведението на два полинома.

Някои от тези функции са реализирани в програмата Zad150.cpp. Полинома ще реализираме чрез специализацията на класа LList с базов тип pol, определен като структура.

```
// Program Zad150.cpp
#include <iostream.h>
#include "L-List.cpp"
struct pol
{int pow;
 double coef;
};
typedef LList<pol> polinom;
//a)
void read(polinom &p)
{char flag;
 do
 {pol ed;
  cout << "pow: "; cin >> ed.pow;
  cout << "coef: "; cin >> ed.coef;
  p.ToEnd(ed);
  cout << "Input y/n: ";
  cin >> flag;
 } while (flag == 'y');
}
// б)
void LList<pol>::print()
{elem<pol> *p = Start;
```

```
 while (p)
 {if (p->inf.coef > 0) cout << " + ";
  else cout << " ";
  cout << p->inf.coef << " x ^ " << p->inf.pow;
  p = p->link;
 }
 cout << endl;
}
// r)
void diff(polinom p, polinom &newp)
{p.IterStart();
 elem<pol> *q = p.Iter();
 while (q)
 {pol ed;
  ed.coef = q->inf.coef * q->inf.pow;
  ed.pow = q->inf.pow - 1;
  newp.ToEnd(ed);
  q = q->link;
 }
}
// e)
void sum(polinom p, polinom q, polinom& r)
{p.IterStart();
 q.IterStart();
 elem<pol> *pp = p.Iter(),
           *qq = q.Iter();
 while (pp && qq)
 if (p->inf.pow > qq->inf.pow)
 {r.ToEnd(pp->inf);
  pp = pp->link;
 }
 else
 if (qq->inf.pow > pp->inf.pow)
 {r.ToEnd(qq->inf);
  qq = qq->link;
 }
```

```
 else
 {pol ed;
  ed.pow = pp->inf.pow;
  ed.coef = pp->inf.coef + qq->inf.coef;
  pp = pp->link;
  qq = qq->link;
  r.ToEnd(ed);
 }
 if (!pp)
  while (qq)
  {r.ToEnd(qq->inf);
   qq = qq->link;
  }
 else
 while (pp)
 {r.ToEnd(pp->inf);
  pp = pp->link;
 }
}
void main()
{polinom p, p1, p2;
 read(p); p.print();
 read(p1); p1.print();
 sum(p, p1, p2);
 p2.print();
 diff(p, p2);
p2.print();
}
```

**Функции от по-висок ред за работа със списъци, представени с една връзка**

***Функция accumulate***

Нека l е списък с елементи от тип T, а op е лявоасоциативна операция от вида:

op: T x T $\longrightarrow$ T

с начална стойност null_val. Ще дефинираме шаблон на фукция от по-висок ред accumulate, чрез който да се реализира натрупване на елементите на списъка l чрез операцията op.

```
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = p->link;
 }
 return s;
}
```

**Задача 151.** Като се използва функцията от по-висок ред accumulate, да се напише програма, която намира стойността на полинома $P_n(x) = (...(a_0x + a_1)x + ... +a_{n-1})x + a_n$ за дадено x по метода на Хорнер.

Полиномът се задава чрез списък от реални числа $a_0$, $a_1$, ..., $a_{n-1}$, $a_n$ и реалното число x. В сила е $P_n(x) = x.P_{n-1}(x) + a_n$. Операцията op ще дефинираме по следния начин:

op: term x coef $\longrightarrow$ x.term + coef

Програма Zad151.cpp решава задачата.

```
// Program Zad151.cpp
#include <iostream.h>
#include "L-List.cpp"
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = p->link;
 }
```

```
 return s;
}
double x;
typedef LList<double> DList;
double op(double term, double coef)
{return x * term + coef;
}
double horner(DList l)
{return accumulate(op, 0.0, l);
}
void readpol(int n, DList & l)
{for (int i = n; i >= 0; i--)
 {cout << "coef: ";
  double coef;
  cin >> coef;
  l.ToEnd(coef);
 }
}
void main()
{DList l;
 int n; // степен на полинома
 cout << "n: ";
 cin >> n;
 readpol(n, l);
 cout << "x= "; cin >> x;
 cout << horner(l) << endl;
}
```

***Функция map***

Нека l е списък с елементи от тип T, а f е едноаргументна функция от вида:

f: T $\longrightarrow$ T

Ще дефинираме шаблон на функция от по-висок ред map за намиране на списък, резултат от прилагането на функцията f над всеки от елементите на l.

```
template <class T>
```

```
LList<T> map(T (*f)(T), LList<T> & l)
{l.IterStart();
 elem<T> *p = l.Iter();
 LList<T> l1;
 while (p)
 {l1.ToEnd(f(p->inf));
  p = p->link;
 }
 return l1;
}
```

Ако е даден списък от реални числа l, след обръщението map(sin, l) се получава списъкът от синусите на елементите на l.

***Функция filter***

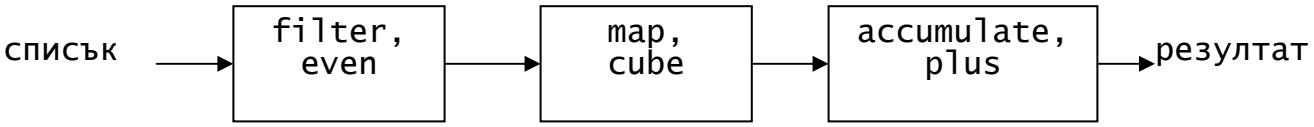
Нека l е списък с елементи от тип T, а pred е предикат. Ще дефинираме шаблон на функция от по-висок ред filter, чрез който се намира списък, състоящ се от онези елементи на l, за които е в сила pred.

```
template <class T>
LList<T> filter(bool (*pred)(T), LList<T> & l)
{LList<T> l1;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {if (pred(p->inf)) l1.ToEnd(p->inf);
  p = p->link;
 }
 return l1;
}
```

Ако l е списък от цели числа, чрез обръщението filter(odd, l), където odd е предикат, установяващ дали аргументът му е нечетно число, се получава списъкът от нечетните елементи на списъка l.

**Задача 152.** Да се напише програма, която намира сумата от кубовете на четните числа на даден списък от цели числа.

За решаването на тази задача ще преминем през следните стъпки:




```
// Program Zad152.cpp
#include <iostream.h>
#include "L-List.cpp"
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = p->link;
 }
 return s;
}
template <class T>
LList<T> map(T (*f)(T), LList<T> &l)
{l.IterStart();
 elem<T> *p = l.Iter();
 LList<T> l1;
 while (p)
 {l1.ToEnd(f(p->inf));
  p = p->link;
 }
 return l1;
}
template <class T>
LList<T> filter(bool(*pred)(T), LList<T> & l)
```

```
{LList<T> l1;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {if (pred(p->inf)) l1.ToEnd(p->inf);
  p = p->link;
 }
 return l1;
}
int plus(int a, int b)
{return a + b;
}
bool even(int x)
{return x%2 == 0;
}
int cube(int x)
{return x*x*x;
}
void main()
{LList<int> m, n;
 int k; cin >> k;
 for (int j = 1; j <= k; j++)
 {cout << "number: ";
  int a;
  cin >> a;
  m.ToEnd(a);
 }
 m.print();
 n = map(cube, filter(even, m));
 n.print();
 cout << accumulate(plus, 0, n) << endl;
}
```

**Задача 153.** Да се напише програма, която създава списък от списъци от реални числа, след което конкатенира списъците, съставящи списъка.

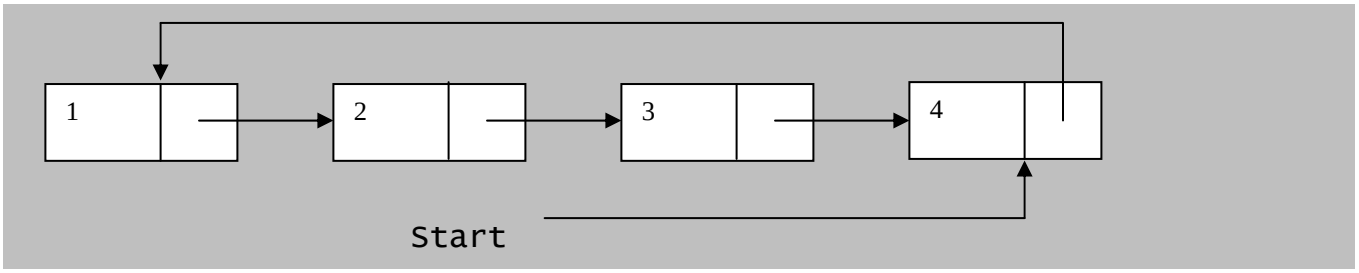
Програма Zad153.cpp решава задачата.

```
// Program Zad153.cpp
#include <iostream.h>
include "L-List.cpp"
typedef LList<double> DList;
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = p->link;
 }
 return s;
}
DList concat(DList l1, DList l2)
{DList l;
 l1.IterStart();
 l2.IterStart();
 elem<double> *p = l1.Iter(),
              *q = l2.Iter();
 while (p)
 {l.ToEnd(p->inf);
  p = p->link;
 }
 while (q)
{l.ToEnd(q->inf);
  q = q->link;
 }
 return l;
}
void main()
{LList<DList> l;
 int n;
 cout << "n= ";
```

```
    cin >> n;
    for (int i = 1; i <= n; i++)
    {DList m;
     cout << "k= "; int k;
     cin >> k;
     for (int j = 1; j <= k; j++)
     {cout << "elem: ";
      int el;
      cin >> el;
      m.ToEnd(el);
     }
     l.ToEnd(m);
    }
    DList l2;
    DList l1 = accumulate(concat, l2, l);
    l1.print();
   }
```

### 15.3.4 Циклични списъци

В редица случаи се налага да се използват разновидности на свързаните списъци. Например, ако искаме операцията включване да се осъществява само в началото или само в края на свързания списък, не е нужно да поддържаме два указателя Start и End към началото и края съответно. В този и в редица други случаи се използват т.нар. **циклични** или **кръгови списъци.** При тях се използва само един указател Start, указващ последния елемент на списъка, както е показано на Фиг. 15. 6.




Фиг. 15.6 Представяне на цикличен списък с четири елемента

### 15.3.4.1 Реализация на цикличен списък

Шаблонът на класа cirlist реализира циклични свързани списъци от тип T.

```
template <class T>
struct elem
{T inf;
 elem *link;
};
template <class T>
class CirList
{public:
  CirList();
  ~CirList();
  CirList(CirList &);
  CirList& operator=(CirList &);
  void print();
  void IterStart(elem<T>* = NULL);
  elem<T>* Iter();
  void ToEnd(T &);
  void Insert(T &);
  void DeleteElem(elem<T>*, T &);
 private:
  elem<T> *Start,
          *Current;
  void DeleteList();
  void CopyList(CirList &);
};
```

Конструкторът по подразбиране и член-функциите на голямата тройка реализират аналогични идеи на тези при свързания списък с една връзка. Забелязваме, че формалните параметри на член-функциите

```
cirlist(cirlist &);
cirlist& operator=(cirlist &);
```

не са const. Това се налага заради функцията CopyList, чийто формален параметър също не е const. Последното пък е свързано с използването на функциите за работа с итератора при обхождането за копиране.

```
template <class T>
CirList<T>::CirList()
{Start = NULL;
}
template <class T>
CirList<T>::~CirList()
{DeleteList();
}
template <class T>
CirList<T>::CirList(CirList<T> & r)
{CopyList(r);
}
template <class T>
CirList<T>& CirList<T>::operator=(CirList<T> & r)
{if (this != &r)
 {DeleteList();
  CopyList(r);
 }
 return *this;
}
```

Тъй като при програмирането на член-функциите на шаблона на класа CirList ще използваме функциите за работа с итератора, отначало ще разгледаме тези функции.

*Установяване на итератора в началото на цикличен списък*

Указателят Start сочи последния елемент на цикличния списък. Елементът, намиращ се след указания от Start (ако Start не е NULL) е началото на списъка. Функцията IterStart установява Current в указан адрес, в NULL – ако списъкът е празен и в началото на списъка в противен случай.

```
template <class T>
void CirList<T>::IterStart(elem<T> *p)
{if (p) Current = p;
```

```
 else
 if (!Start) Current = NULL;
 else Current = Start->link;
}
```

*Преместване на итератора в следваща позиция*

Член-функцията Iter() осигурява преместване на Current в следващата позиция. При достигане до Start (указващ последния елемент на кръговия списък), Current става NULL.

```
template <class T>
elem<T>* CirList<T>::Iter()
{if (!Current) return NULL;
 elem<T> *p = Current;
 if (Current == Start) Current = NULL;
  else Current = Current->link;
  return p;
 }
```

Помощните член-функции за изтриване и копиране на цикличен списък DeleteList и CopyList съответно използват функциите за работа с итератора.

*Изтриване на списък*

```
template <class T>
void CirList<T>::DeleteList()
{IterStart();
 elem<T> *p = Iter();
 while (p)
 {delete p;
  p = Iter();
 }
}
```

*Копиране на списък*

```
template <class T>
void CirList<T>::CopyList(CirList & r)
```

```
{Start = NULL;
 r.IterStart();
 elem<T> *p = r.Iter();
 while (p)
 {ToEnd(p->inf);
  p = r.Iter();
 }
}
```

*Включване на елемент*

Ще реализираме две член-функции за включване на елемент в цикличен списък. Функцията Insert включва указан като формален параметър елемент след последния (пред първия) елемент на списъка, а функцията ToEnd включва указания елемент като последен елемент на списъка.

```
template <class T>
void CirList<T>::Insert(T & x)
{elem<T> *p = new elem<T>;
 p->inf = x;
 if (Start) p->link = Start->link;
 else Start = p;
 Start->link = p;
}
template <class T>
void CirList<T>::ToEnd(T & x)
{Insert(x);
 Start = Start->link;
}
```

*Изтриване на елемент*

Член-функцията Delete изтрива указания от указателя p елемент и го запомня в параметъра x.

```
template <class T>
void CirList<T>::DeleteElem(elem<T>* p, T & x)
{x = p->inf;
 if (Start != Start->link)
 {elem<T> *q = Start;
```

```
  while (q->link != p) q = q->link;
  q->link = p->link;
  if (p == Start) Start = q;
  delete p;
 }
 else
 {Start = NULL;
  delete p;
 }
}
```

*Извеждане на цикличен списък*

```
template <class T>
void CirList<T>::print()
{IterStart();
 elem<T> *p = Iter();
 while (p)
 {cout << p->inf << " ";
  p = Iter();
 }
 cout << endl;
}
```

Ще приложим този шаблон на клас в следната задача.

**Задача 154.** Дадени са естествените числа n и m. Предполага се, че m човека са наредени в кръг и всеки от тях е получил пореден номер от 1 до m (броейки в посока обратна на часовниковата стрелка). След това, започвайки от първия и също в посока обратна на часовниковата стрелка се отброява n-тия човек и се отстранява от кръга. Отново започвайки от следващия човек (n+1-вия) се отброява отново n-тия човек и се отстранява. Този процес продължава до отстраняване на всички от кръга. Да се напише програма, която извежда номерата на отстранените в реда на отстраняване.

Програма Zad154.cpp решава задачата.

```
// Program Zad154.cpp
#include <iostream.h>
#include “CirList.cpp”
typedef CirList<int> IntCir;
void create(int m, IntCir &l)
{for (int i = 1; i <= m; i++)
 l.ToEnd(i);
}
void josiff(int n, IntCir l)
{l.IterStart();
 elem<int> *p = l.Iter(), *q;
 while (p != p->link)
 {q = p;
  for (int i = 1; i <= n-1; i++)
    q = q->link;
  p = q->link;
  int x; l.DeleteElem(q, x);
  cout << x << " ";
 }
 cout << p->inf << endl;
}
void main()
{IntCir l;
 create(10, l);
 l.print();
 josiff(3,  l);
}
```

### 15.3.4.2 Функции от по-висок ред за работа със циклични списъци

**Задача 155.** Да се напише шаблон на функцията от по-висок ред accumulate за циклични списъци.

```
template <class T>
T accumulate(T (*op)(T, T), T null_val, CirList<T>& l)
{T s = null_val;
```

```
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = l.Iter(); //използването на p=p->list ще доведе до зацикляне
 }
 return s;
}
```

**Задача 156.** Да се напише шаблон на функцията от по-висок ред map за циклични списъци.

```
template <class T>
CirList<T> map(T (*f)(T), CirList<T> &l)
{l.IterStart();
 elem<T> *p = l.Iter();
 CirList<T> l1;
 while (p)
 {l1.ToEnd(f(p->inf));
  p = l.Iter();
 }
 return l1;
}
```

**Задача 157.** Да се напише шаблон на функцията от по-висок ред filter за циклични списъци.

```
template <class T>
CirList<T> filter(bool (*pred)(T), CirList<T> &l)
{l.IterStart();
 elem<T> *p = l.Iter();
 CirList<T> l1;
 while (p)
 {if (pred(p->inf)) l1.ToEnd(p->inf);
  p = l.Iter();
 }
```

```
 return l1;
}
```

**Задача 158.** Да се напише програма, която създава цикличен списък, изграден от циклични списъци от числа, след което конструира цикличен списък, резултат от конкатенацията на цикличните списъци, съставящи дадения списък.

Програма Zad158.cpp решава задачата. Тя използва функцията от по-висок ред accumulate. За да се избегне зациклянето, в оператора за цикъл е използвана член-функцията Iter().

```
// Program Zad158.cpp
#include <iostream.h>
#include "CirList.cpp"
template <class T>
T accumulate(T (*op)(T, T), T null_val, CirList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = l.Iter(); //използването на p=p->list ще доведе до зацикляне
 }
 return s;
}
typedef CirList<int> IntList;
typedef CirList<IntList> IntListList;
// конкатениране на циклични списъци
IntList concat(IntList l1, IntList l2)
{IntList l;
 l1.IterStart();
 l2.IterStart();
 elem<int> *p = l1.Iter(),
           *q = l2.Iter();
 while (p)
```

```
 {l.ToEnd(p->inf);
  p = l1.Iter();
 }
 while (q)
 {l.ToEnd(q->inf);
  q = l2.Iter();
 }
 return l;
}
// създаване на цикличен списък от n циклични списъка
void create(int n, IntListList &l)
{for (int i = 1; i <= n; i++)
 {IntList m;
  cout << "k= ";
  int k;
  cin >> k;
  for (int j = 1; j <= k; j++)
  {cout << "x: ";
   int x;
   cin >> x;
   m.ToEnd(x);
  }
  l.ToEnd(m);
 }
}
void main()
{int n;
 cout << "n= ";
 cin >> n;
 IntListList l;
 create(n, l);
 IntList l2;
 IntList l1 = accumulate(concat, l2, l);
 l1.print();
}
```

### 15.3.5 Реализация на свързан списък с две връзки

Шаблонът на клас DLList реализира това представяне на свързан списък.

```
template <class T>
struct elem
{T inf;
 elem *pred,
      *succ;
};
template <class T>
class DLList
{public:
  DLList();
  ~DLList();
  DLList(DLList const&);
  DLList& operator=(DLList const &);
  void print();
  void print_reverse();
  void IterStart(elem<T>* = NULL);
  void IterEnd(elem<T>* = NULL);
  elem<T>* IterSucc();
  elem<T>* IterPred();
  void ToEnd(T const &);
  void DeleteElem(elem<T> *, T &);
  int len();
 private:
  elem<T> *Start,
          *End,
          *CurrentS,
          *CurrentE;
  void DeleteList();
  void CopyList(DLList const &);
};
```

Конструктора и член-функциите на голямата тройка няма да коментираме. Реализират аналогични идеи на тези при свързаното представяне с една връзка.

```
template <class T>
DLList<T>::DLList()
{Start = NULL;
 End = NULL;
}
template <class T>
DLList<T>::~DLList()
{DeleteList();
}
template <class T>
DLList<T>::DLList(DLList<T> const& r)
{CopyList(r);
}
template <class T>
DLList<T>& DLList<T>::operator=(DLList<T> const & r)
{if (this != &r)
 {DeleteList();
  CopyList(r);
 }
 return *this;
}
```

Следващите две функции са помощни и реализират изтриване на свързан списък и копиране на свързан списък на друго място в паметта. Използват се за реализиране на голямата тройка и затова са капсолирани в private-частта на класа.

*Изтриване на списък*

```
template <class T>
void DLList<T>::DeleteList()
{elem<T> *p = Start;
 while (p)
 {Start = Start->succ;
  delete p;
```

```
  p = Start;
 }
 End = NULL;
}
```

*Копиране на списък*

```
template <class T>
void DLList<T>::CopyList(DLList<T> const & r)
{Start = End = NULL;
 if (r.Start)
 {elem<T> *p = r.Start;
  while (p)
  {ToEnd(p->inf);
   p = p->succ;
  }
 }
}
```

*Извеждане на елементите на списък*

Извеждането на елементите на свързан списък се реализира чрез последователното им обхождане, без разрушаването на списъка. При обхождане от Start, списъкът се извежда от началото му, а при обхождане, започващо от End, извеждането е в обратен ред.

*- извеждане от началото към края*

```
template <class T>
void DLList<T>::print()
{elem<T> *p = Start;
 while (p)
 {cout << p->inf << " ";
  p = p->succ;
 }
 cout << endl;
}
```

*- извеждане от края към началото*

```
template <class T>
void DLList<T>::print()
```

```
{elem<T> *p = End;
 while (p)
 {cout << p->inf << " ";
  p = p->pred;
 }
 cout << endl;
}
```

Следващите четири член-функции реализират операциите с итераторите CurrentS и CurrentE.

*Установяване на итератора в началото на списъка*

```
template <class T>
void DLList<T>::IterStart(elem<T> *p)
{if (p) CurrentS = p;
 else CurrentS = Start;
}
```

*Установяване на итератора в края на списъка*

```
template <class T>
void DLList<T>::IterEnd(elem<T> *p)
{if (p) CurrentE = p;
 else CurrentE = End;
}
```

*Преместване на итератора в следващата позиция*

```
template <class T>
elem<T>* DLList<T>::IterSucc()
{elem<T> *p = CurrentS;
 if (CurrentS) CurrentS = CurrentS->succ;
 return p;
}
```

*Преместване на итератора в предходната позиция*

```
template <class T>
elem<T>* DLList<T>::IterPred()
{elem<T> *p = CurrentE;
```

```
 if (CurrentE) CurrentE = CurrentE->pred;
 return p;
}
```

*Включване на елемент в списък*

Включването на елемент в края на свързан списък ще реализираме чрез член-функцията ToEnd.

```
template <class T>
void DLList<T>::ToEnd(T const & x)
{elem<T>* p = End;
 End = new elem<T>;
 End->inf = x;
 End->succ = NULL;
 if (p) p->succ = End;
 else Start = End;
 End->pred = p;
}
```

*Изключване на указан елемент от списък*

Изключването на указан елемент в свързан списък ще реализираме чрез следната член-функция:

```
template <class T>
void DLList<T>::DeleteElem(elem<T> *p, T &x)
{x = p->inf;
 if (Strat == End)
 {Start = End = NULL;
 }
 else
 if (p == Start)
 {Start = Start->succ;
  Start->pred = NULL;
 }
 else
 if (p == End)
 {End = p->pred;
  End->succ = NULL;
```

```
 }
 else
 {p->pred->succ = p->succ;
  p->succ->pred = p->pred;
 }
 delete p;
}
```

*Дължина на свързан списък*

Намирането дължината на свързан списък се осъществява по същия начин, като при представянето с една връзка.

```
template <class T>
int DLList<T>::len()
{int n = 0;
 elem<T> *p = Start;
 while (p)
 {n++;
  p = p->succ;
 }
 return n;
}
```

## Задачи върху свързан списък с две връзки

Шаблонът на класа DLList записваме във файла DL-List.cpp.

**Задача 159.** Свързан списък, съдържащ 2n цели числа $a_1$, $a_2$,...,$a_{2n-1}$, $a_{2n}$, е представен чрез две връзки. Да се напише функция, която намира:

а) $S = a_1.a_{2n} + a_2.a_{2n-1} + a_n.a_{n+1}$

б) $M = \max\{\min\{a_1, a_{2n}\}, \min\{a_2, a_{2n-1}\}, ..., \min\{a_n, a_{n+1}\}\}$

в) $M = \max\{\min\{a_1, a_2, ..., a_n\}, \min\{a_{n+1}, a_{n+2}, a_{2n}\}\}$.

Програма Zad159.cpp решава задачата.

```
// Program Zad159.cpp
#include <iostream.h>
#include "DL-List.cpp"
```

```
typedef DLList<int> IntList;
// създаване на свързан списък с две връзки
void CreateList(int n, IntList &l)
{for (int i = 1; i <= 2*n; i++)
{cout << "Enter a number: ";
 int x;
 cin >> x;
 l.ToEnd(x);
}
}
// a)
int sum(IntList &l)
{l.IterStart();
 elem<int> *p = l.IterSucc();
 l.IterEnd();
 elem<int> *q = l.IterPred();
 int n = l.len()/2;
 int s = 0;
 for (int i = 1; i <= n; i++)
 {s = s + p->inf * q->inf;
  p = l.IterSucc();
  q = l.IterPred();
 }
 return s;
}
// б)
int maximin(IntList &l)
{int max;
 l.IterStart();
 elem<int> *p = l.IterSucc();
 l.IterEnd();
 elem<int> *q = l.IterPred();
 int n = l.len()/2;
 if (p->inf <= q->inf) max = p->inf;
 else max = q->inf;
 for (int i = 1; i <= n-1; i++)
```

```
 {p = l.IterSucc();
  q = l.IterPred();
  int min;
  if (p->inf <= q->inf) min = p->inf;
  else min = q->inf;
  if (min > max) max = min;
 }
 return max;
}
void main()
{IntList l;
 cout << "n= ";
 int n;
 cin >> n;
 CreateList(n, l);
 l.print();
 cout << “sum: “ << sum(l) << endl;
 cout << “maximin: “ << maximin(l) << endl;
}
```

**Задача 160.** Даден е свързан списък, съдържащ 2n цели числа $a_1$, $a_2$, ..., $a_{2n-1}$, $a_{2n}$, представен чрез две връзки така, че първите му n елемента са сортирани във възходящ, а вторите n елемента – в низходящ ред. Да се напише функция, която сортира във възходящ ред елементите на списъка.

Функцията Sort решава задачата. Тя реализира следната идея. Поставя указатели p и q към началото и към края съответно. Докато p и q са различни, по-малкият от сочените от p и q елементи се включва в нов списък, след което се прескача.

```
IntList Sort(IntList &l)
{l.IterStart();
 elem<int> *p = l.IterSucc();
 l.IterEnd();
 elem<int> *q = l.IterPred();
 IntList l1;
```

```
 while (p != q)
 if (p->inf <= q->inf)
 {l1.ToEnd(p->inf);
  p = l.IterSucc();
 }
 else
 {l1.ToEnd(q->inf);
  q = l.IterPred();
 }
 l1.ToEnd(p->inf);
 return l1;
}
```

## Задачи

**Задача 1.** Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елемент принадлежи на стек.

**Задача 2.** Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на стек от числа (думи) са наредени във възходящ (лексикографски) ред.

**Задача 3.** Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на стек от числа са различни.

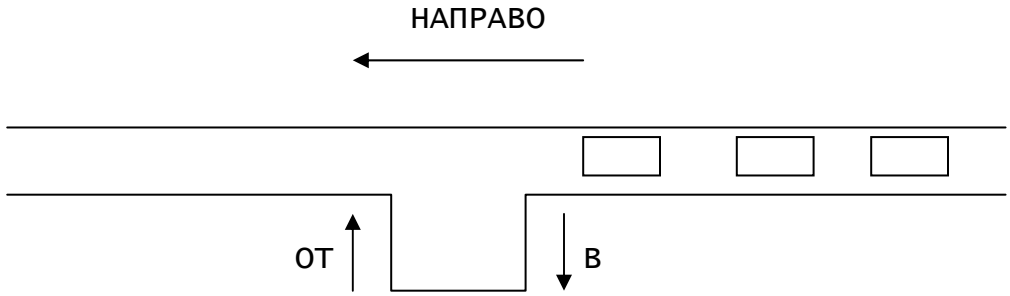
**Задача 4.** Даден е стек от цели числа. Да се напише булева функция, която установява дали елементите на стека могат да се пренаредят така, че разликата на всеки два съседни елемента да е наредена по намаляване. Ако за дадения стек това е възможно, да се пренаредят елементите на стека, след което се изведат.

**Задача 5.** В стек е записан без грешка израз от вида:

```
<израз> ::= <цифра>|
            s(<израз>)|
            p(<израз>)
```

където s(x) и p(x) намират x+1 и x-1, съответно (s(9) се приема за 0, а p(0) – за 9). Като се използва стек да се намери стойността на израза.

**Задача 6.** Железопътен сортировъчен възел е построен по следния начин:




разклонение

В дясната страна са събрани n черни и n бели вагона. В разклонението могат да се поместят всички вагони. Като се използват трите операции В, ОТ и НАПРАВО, да се съберат вагоните в лявата страна така, че да се редуват вагоните от двата вида. За целта да се използват не повече от 3n-1 операции В, ОТ и НАПРАВО.

**Задача 7.** Железопътен сортировъчен възел е построен по начина от предната задача. В дясната част са разположени n различни вагона, номерирани с 1, 2, ..., n. Да се напише програма, която като използва операците В, ОТ и НАПРАВО, намира всички възможни начини за подреждане на вагоните.

**Задача 8.** Да се напише програма, която намира колко различни пермутации на елементите 1, 2, ..., n могат да се конструират като се използва стекът от задача 6 и операциите В, ОТ и НАПРАВО.

**Задача 9.** Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елемент принадлежи на опашка.

**Задача 10.** Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на опашка от числа (думи) са наредени във възходящ (лексикографски) ред.

**Задача 11.** Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на опашка от числа са различни.

**Задача 12.** Дадени са две опашки от цели числа p и q. Да се дефинира функция, която намира опашка от цели числа, съдържаща елементите, които:

а) принадлежат на поне една от опашките p или q;

б) принадлежат едновременно и на p, и на q;

в) принадлежат на p, но не принадлежат на q;

г) принадлежат на една от опашките, но не принадлежат на другата.

**Задача 13.** Да се дефинира функцията от по-висок ред accumulate за стек (опашка).

**Задача 14.** Да се дефинира функцията от по-висок ред map за стек (опашка).

**Задача 15.** Да се дефинира функцията от по-висок ред filter за стек (опашка).

**Задача 16.** Символен низ съдържа правилен текст от вида:

<текст> ::= <интервал> | <елемент><текст>

<елемент> ::= <буква> | (<текст>).

Като се използват опашка и/или стек, да се напише функция, която за всяка двойка съответстващи си отваряща и затваряща скоби извежда позициите им в текста, подредени по нарастване на позицията на отварящите скоби.

(Например, за текста A+(TP-F(X)*(B-C)) резултатът е 3 17; 8 10; 12 16.)

**Задача 17.** Символен низ съдържа правилен текст от вида:

<текст> ::= <интервал> | <елемент><текст>

<елемент> ::= <буква> | (<текст>).

Като се използват опашка и/или стек, да се напише функция, която за всяка двойка съответстващи си отваряща и затваряща скоби извежда позициите им в текста, подредени по нарастване на позицията на затварящите скоби. (Например, за текста A+(TP-F(X)*(B-C)) резултатът е 8 10; 12 16; 3 17.)

**Задача 18.** Даден е списък от четен брой символи. Да се напише булева функция, която определя дали елементите от първата половина съвпадат с елементите от втората половина на списъка. Елементите на списъка да се сканират отляво надясно.

**Задача 19.** Да се напише функция, която изтрива от списъка l:

а) първия отрицателен елемент, ако такъв съществува;

б) всички отрицателни елементи.

**Задача 20.** Да се напише програма, която създава свързан списък, съдържащ информация за студентите от една група. Програмата да може да:

а) добавя данни за нов студент;

б) изтрива данни за студент;

в) търси данни за студент;

г) сортира по някаква данна елементите на списъка.

**Задача 21.** Да се напише програма, която създава свързан списък, елементите, на който са свързани списъци от вида, описан в предната задача. Сортира всеки от свързаните списъци по успеха на студентите и извежда получения списък от списъци.

**Задача 22.** Даден е свързан списък с 2n елемента, представен с две връзки. Да се напише функция, която проверява дали елементите на списъка са симетрични относно средата на списъка.

**Задача 23.** Да се напише програма, която създава цикличен списък от циклични списъци от символи. Да се конструира цикличен списък от думите, получени след конкатенирането на символите на всеки от цикличните списъци. Да се конструира изречение, получено след конкатенирането на всяка от думите на цикличния списък.

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. А. Берстисс, Структуры данных, М. Статистика, 1974.
3. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
4. Л. Амерал, Алгоритми и структури от данни в C++, София, ИК СОФТЕХ, 2001.
5. М. Тодорова, Програмиране на Паскал, София, Полипринт, 1993.

# 16

# Йерархични структури от данни двоично дърво и граф

## 16.1 Двоично дърво

### 16.1.1 Дефиниране на двоично дърво

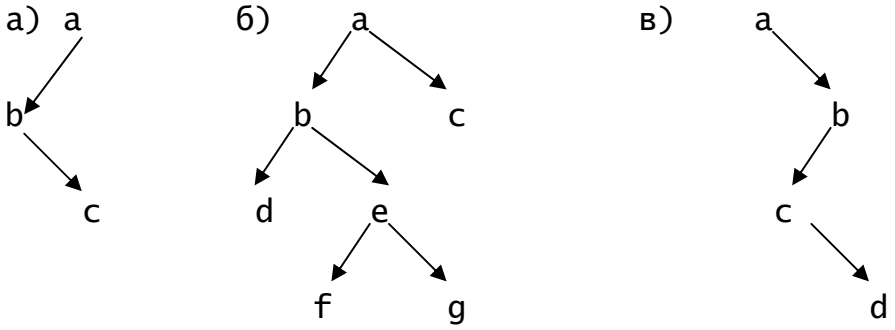
**Логическо описание**

**Двоично дърво от тип T** е структура от данни, която е или празна, или е образувана от

- данна от тип T, наречена **корен** (връх, възел) на двоичното дърво от тип T;
- двоично дърво от тип T, наречено **ляво поддърво** на двоичното дърво от тип T (ЛПД);
- двоично дърво от тип T, наречено **дясно поддърво** на двоичното дърво от тип T (ДПД).

**Примери:**

Нека a, b, c, d, e, f и g са данни от тип T. Тогава следните графични представяния определят двоични дървета от тип T.



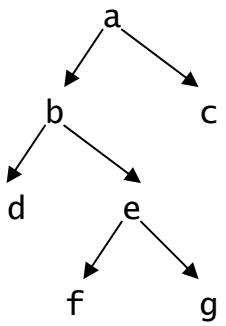
Посоката на линиите, свързващи върховете с поддървета, позволява да се различи ляво от дясно поддърво. Двоичните дървета

са различни. В единия случай дясното поддърво е празно, а в другия - дясното дърво не е празно.

*Някои определения*

**Листо** - това е връх с празни поддървета.

**Пример:** Върховете d, c, f и g на двоичното дърво




са листа.

Върховете, които не съвпадат с корена и листата, се наричат **вътрешни върхове**.

**Пример:** b и e, от примера по-горе, са вътрешни върхове на двоичното дърво от тип T.

Всяко поддърво се нарича **наследник (син)** по отношение на своето дърво и **родител (баща)** по отношение на своите поддървета.

На всеки връх може да се съпостави "**ниво**". Приемаме, че коренът има ниво 1 (или 0) и ако един връх има ниво i и има наследници, то те имат ниво i+1. Така **нивото на връх** е всъщност броя на върховете, които трябва да бъдат обходени като се започне от корена и се стигне до върха. Най-голямото ниво на двоично дърво, се нарича негова **височина** (**дълбочина**).

Над структурата от данни двоично дърво са възможни следните операции:

*- достъп до връх*

Възможен е пряк достъп до корена и непряк достъп до всеки от останалите върхове на двоичното дърво.

*- включване и изключване на връх*

Включването и изключването са възможни в произволно място на двоичното дърво, но в резултат трябва да се получи двоично дърво от тип Т.

*Обхождане на двоично дърво*

Обхождането на двоично дърво дава метод, позволяващ да се осъществи достъп до всеки връх на дървото един единствен път.

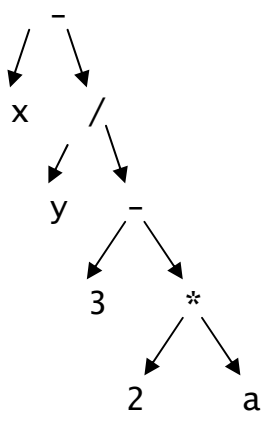
Осъществява се чрез изпълнение на следните три действия в някаква последователност:

- обработка на корена;
- обхождане на лявото поддърво;
- обхождане на дясното поддърво,

т.е. процесът на обхождане е рекурсивен. Съществуват шест различни начина за обхождане на двоично дърво: ЛКД (**смесено обхождане**), КЛД (**низходящо обхождане**), ЛДК (**възходящо обхождане**), ДКЛ, КДЛ и ДЛК, където К - означава корен, Л - ляво поддърво, Д - дясно поддърво. Обхождането ЛКД например означава, че първо се обхожда лявото поддърво, след него се обработва коренът и накрая се обхожда дясното поддърво.

Всеки аритметичен израз може да се представи чрез двоично дърво, в листата на което са операндите, а във вътрешните върхове и корена - операциите.

**Пример:** Изразът x-y/(3-2*a) може да се представи чрез двоичното дърво:




Възходящият обход на двоично дърво, представящо аритметичен израз, дава обратния полски запис на аритметичния израз. Смесеният обход на двоично дърво, представящо аритметичен израз, дава общоприетия

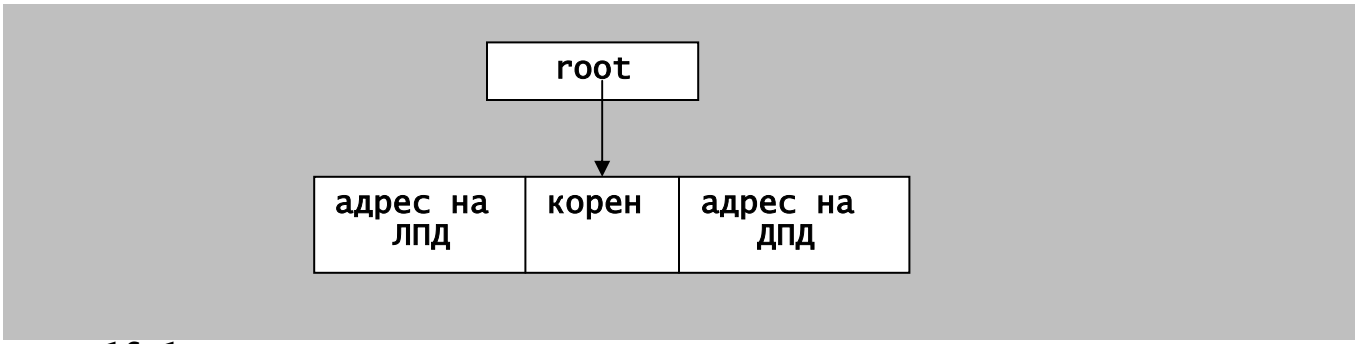
(инфиксен) запис на аритметичния израз, но без скобите, а низходящият обход на двоично дърво, представящо аритметичен израз, дава представяне, което се нарича **прав полски запис**.

**Физическо представяне**

Използват се главно два начина за физическо представяне на двоично дърво от тип T - **свързано** и **верижно**.

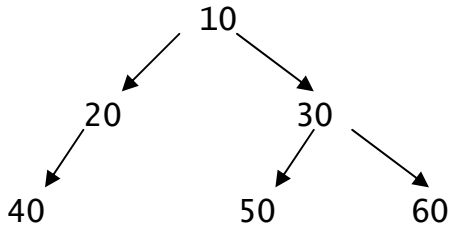
*Свързано представяне*

Реализира се чрез указател към кутия с три полета: информационно, съдържащо стойността на корена и две адресни, съдържащи представянията на ЛПД и ДПД съответно (Фиг. 16.1).

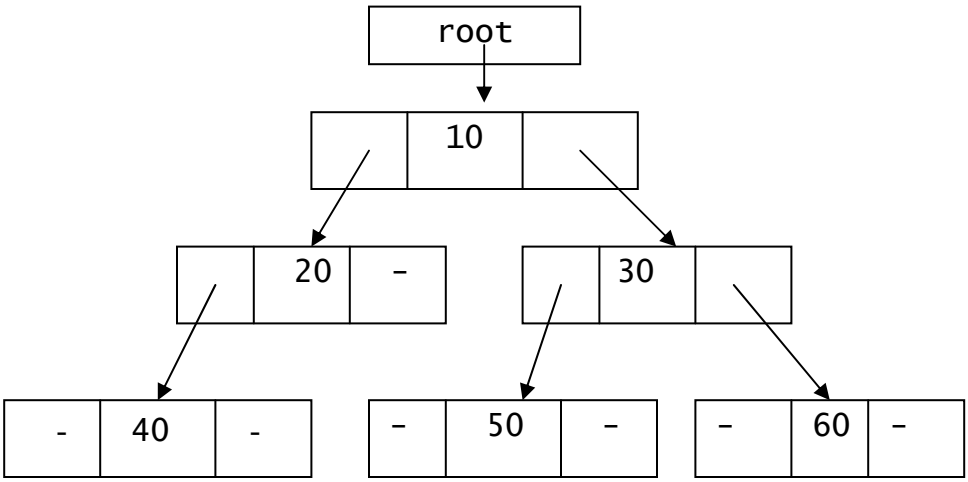



Фиг. 16.1 Свързано представяне на двоично дърво

*Пример:* Двоичното дърво



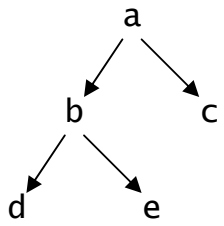

се представя по следния начин:



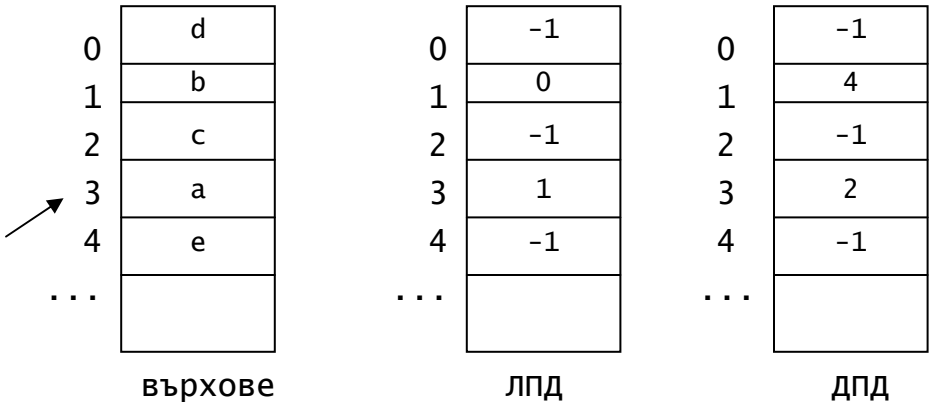
*Верижно представяне*

При това представяне се използват три масива - за върховете на дървото, за адресите на лявото и за адресите на дясното поддърво. Ролята на адреси се изпълнява от индекси. i-ят елемент на масива "върхове" съдържа стойността на връх на двоичното дърво, i-ят елемент на ЛПД съдържа адреса на лявото поддърво на поддървото с корен i-я елемент, а i-ят елемент на ДПД - адреса на дясното му поддърво. Поддържа се указател, който съдържа адреса на корена.

**Пример:** Верижното представяне на двоичното дърво от тип T




се представя по следния начин:




Указателят към корена е 3. Празното дърво е представено чрез –1.

В следващите разглеждания ще използваме свързаното представяне на двоично дърво.

## 16.1.2 Реализация на свързаното представяне

Тройната кутия, съдържаща корена и адресите на лявото и дясното поддървета, представяме чрез следния шаблон на структурата node.

```
template <class T>
struct node
```

```
{T inf;
 node *Left,
      *Right;
};
```

Двоичното дърво ще представим чрез указател към тройна кутия от вида, описан по-горе. Ще го реализираме чрез шаблона на класа tree

```
template <class T>
class tree
{public:
 tree();
 ~tree();
 tree(tree const&);
 tree& operator=(tree const&);
 bool empty() const;
 T RootTree() const;
 tree LeftTree() const;
 tree RightTree() const;
 void Create3(T, tree<T>, tree<T>);
 void print() const
 {pr(root);
  cout << endl;
 }
 void Create()
 {CreateTree(root);
 }
 private:
  node<T> *root;
  void DeleteTree(node<T>* &) const;
  void Copy(node<T> * &, node<T>* const&) const;
  void CopyTree(tree<T> const&);
  void pr(const node<T> *) const;
  void CreateTree(node<T> * &) const;
};
```

Конструкторът и функциите на голямата тройка реализират познати идеи.

```
template <class T>
```

```
tree<T>::tree()
{root = NULL;
}
template <class T>
tree<T>::~tree()
{DeleteTree(root);
}
template <class T>
tree<T>::tree(tree<T> const& r)
{CopyTree(r);
}
template <class T>
tree<T>& tree<T>::operator=(tree<T> const& r)
{if (this != &r)
{DeleteTree(root);
 CopyTree(r);
}
 return *this;
}
```

За реализирането им използваме член-функциите DeleteTree, CopyTree и Copy на шаблона на класа tree.

```
template <class T>
void tree<T>::DeleteTree(node<T>* &p)const
{if (p)
 {DeleteTree(p->Left);
  DeleteTree(p->Right);
  delete p;
  p = NULL;
 }
}
template <class T>
void tree<T>::CopyTree(tree<T> const& r)
{Copy(root, r.root);
}
template <class T>
void tree<T>::Copy(node<T> * & pos, node<T>* const & r) const
```

```
{pos = NULL;
 if (r)
 {pos = new node<T>;
  pos->inf = r->inf;
  Copy(pos->Left, r->Left);
  Copy(pos->Right, r->Right);
 }
}
```

Използването на допълнителния параметър от тип node<T>* в DeleteTree и Copy е заради рекурсията.

Проверката дали двоично дърво е празно се осъществява чрез булевата член-функция empty() на шаблона.

```
template <class T>
bool tree<T>::empty()const
{return root == NULL;
}
```

Достъпът до корена, до лявото и до дясното поддърво на дадено двоично дърво се осъществява чрез член-функциите: RootTree(), LeftTree() и RightTree().

```
template <class T>
T tree<T>::RootTree()const
{return root->inf;
}
template <class T>
tree<T> tree<T>::LeftTree() const
{tree<T> t;
 Copy(t.root, root->Left);
 return t;
}
template <class T>
tree<T> tree<T>::RightTree() const
{tree<T> t;
 Copy(t.root, root->Right);
 return t;
}
```

Извеждането на елементите на двоично дърво става чрез член-функцията print(). Тъй като реализацията й е рекурсивна, print() използва помощната член-функция pr.

```
template <class T>
void tree<T>::pr(const node<T>*p) const
{if (p)
 {pr(p->Left);
  cout << p->inf << " " ;
  pr(p->Right);
 }
}
```

Следните две член-функции създават двоично дърво. Функцията Create3 създава двоично дърво по дадени корен, ляво и дясно поддървета.

```
template <class T>
void tree<T>::Create3(T x, tree<T> l, tree<T> r)
{root = new node<T>;
 root->inf = x;
 Copy(root->Left, l.root);
 Copy(root->Right, r.root);
}
```

Член-функцията Create създава произволно двоично дърво. Тя използва капсолираната член-функция CreateTree, дефинирана рекурсивно по следния начин:

```
template <class T>
void tree<T>::CreateTree(node<T> * & pos) const
{T x; char c;
 cout << "root: ";
 cin >> x;
 pos = new node<T>;
 pos->inf = x;
 pos->Left = NULL;
 pos->Right = NULL;
 cout << "Left Tree of: " << x << "  y/n? ";
 cin >> c;
 if (c == 'y') CreateTree(pos->Left);
```

```
 cout << "Right Tree of: " << x << " y/n? ";
 cin >> c;
 if (c == 'y') CreateTree(pos->Right);
}
```

Записваме този шаблон във файла Tree.cpp и ще го демонстрираме чрез някои задачи.

## Задачи върху двоично дърво

**Задача 161.** Да се напише програма, която създава двоично дърво от цели числа. Намира и извежда корена, лявото и дясното му поддървета. Конструира и извежда двоично дърво с корен, съвпадащ с корена на първоначалното двоично дърво, с ляво поддърво – дясното поддърво на първоначалното и дясно поддърво – лявото поддърво на първоначалното двоично дърво.

Програма Zad161.cpp решава задачата.

```
// Program Zad161.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
void main()
{IntTree t;
 t.Create(); // създаване на двоично дърво
 t.print(); // извеждане на двоично дърво
 IntTree t1 = t.LeftTree(),   // намиране на ЛПД
         t2 = t.RightTree();  // намиране на ДПД
 int x = t.RootTree();  // намиране на корена
 cout << "Root: " << x << endl;
 cout << "LeftTree: \n";
 t1.print();
 cout << "RightTree: \n";
 t2.print();
 IntTree t3;
 t3.Create3(x, t2, t1); // генериране на новото дърво
```

```
 t3.print();
}
```

**Задача 162.** Да се напише функция, която увеличава всеки от върховете на двоично дърво от цели числа с дадено цяло число.

Програма Zad162.cpp решава задачата.

```
// Program Zad162.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
IntTree AddElem(int a, IntTree const& t)
{IntTree t1;
 if (!t.empty())
 t1.Create3(t.RootTree() + a,
            AddElem(a, t.LeftTree()),
            AddElem(a, t.RightTree()));
 return t1;
 }
void main()
{IntTree t;
 t.Create();
 t.print();
 cout << "Number: ";
 int a; cin >> a;
 AddElem(a, t).print();
}
```

**Задача 163.** Да се дефинира функция от по-висок ред map, прилагаща едноаргументната функция f към всеки от елементите на дадено двоично дърво.

В програма Zad163.cpp е дадена дефиницията на функцията map и използването ѝ за увеличаване с 1 на всеки от елементите на двоично дърво от тип int, а също за прилагане на функцията “факториел” към всеки връх на дадено двоично дърво.

```
// Program Zad163.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
template <class T>
tree<T> map(T (*f)(T), tree<T> const &t)
{tree<T> t1;
 if(!t.empty())
 t1.Create3(f(t.RootTree()), map(f, t.LeftTree()),
                            map(f, t.RightTree()));
 return t1;
}
int f(int x)
{return x + 1;
}
int g(int x)
{if (x == 0) return 1;
 return x * g(x - 1);
}
void main()
{IntTree t;
 t.Create();
 t.print();
 map(f, t).print();
 map(g, t).print();
}
```

**Задача 164.** Да се дефинира функция от по-висок ред accumulate, прилагаща бинарната лявоасоциативна операция op над елементите на дадено двоично дърво в реда ЛКД. Да се използва accumulate за намиране на сумата и произведението на елементите на дадено двоично дърво от тип int.

```
// Program Zad164.cpp
#include <iostream.h>
#include "Tree.cpp"
```

```
typedef tree<int> IntTree;
template <class T>
T accumulate(T (*op)(T, T), T null_val, tree<T> const &t)
{if (!t.empty())
return op(op(accumulate(op,null_val,t.LeftTree()), t.RootTree()),
          accumulate(op, null_val, t.RightTree()));
return null_val;
}
int sum(int x, int y)
{return x + y;
}
int po(int x, int y)
{return x * y;
}
void main()
{IntTree t;
 t.Create();
 t.print();
 cout << accumulate(sum, 0, t) << endl;
 cout << accumulate(po, 1, t) << endl;
}
```

**Задача 165.** Да се дефинира булева функция equal, която установява дали двоичните дървета t1 и t2 от тип T, са равни.

```
template <class T>
bool equal(tree<T> const &t1, tree<T> const &t2)
{if (t1.empty() && t2.empty()) return true;
 if (t1.empty() && !t2.empty() ||
     !t1.empty() && t2.empty()) return false;
 if (t1.RootTree() != t2.RootTree()) return false;
 return equal(t1.LeftTree(), t2.LeftTree()) &&
        equal(t1.RightTree(), t2.RightTree());
}
```

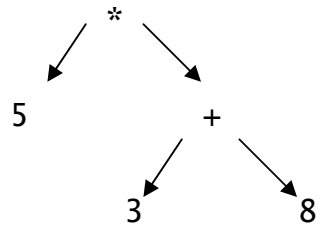
**Задача 166.** Да се напише функция depth, която намира дълбочината на двоично дърво от тип T, т.е. броя на върховете в най-дългия път от корена до листо.

```
template <class T>
int depth(tree<T> const&t)
{if (t.empty()) return 0;
 int n, m;
 n = depth(t.LeftTree());
 m = depth(t.RightTree());
 if (n > m) return n + 1;
 return m + 1;
}
```

**Задача 167.** Формулата

```
<формула> ::= <терминал>|
              (<формула><знак><формула>)
<знак> ::= +|-|*|/
<терминал> ::= 0|1| ...|9
```

може да се представи във вид на двоично дърво от тип char съгласно следното правило: формула състояща се от един терминал се представя чрез двоично дърво с един връх - цифрата; формула от вида (f1 s f2) се представя чрез двоично дърво, коренът на което е знака s, ЛПД съответства на формулата f1, ДПД - на формулата f2. Например, двоичното дърво




представя формулата (5 * (3 + 8))

Да се напише рекурсивна функция или процедура, която:

а) създава двоично дърво, представящо формула от горния вид;

б) проверява, явява ли се двоично дърво, дърво на формула;

в) намира стойността на формула, представена с двоично дърво;

г) извежда двоично дърво във вид, съответстващ на формула.

Програма Zad167.cpp решава задачата. Дървото създаваме чрез член-функцията Create на шаблона на класа tree.

```
// Program Zad167.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<char> CharTree;
// б)
bool IsForm(CharTree const& t)
{if (t.empty()) return false;
 char c = t.RootTree();
 if (c >= '0' && c <= '9')
    return t.LeftTree().empty() && t.RightTree().empty();
 if (c != '+' && c != '-' &&
     c != '*' && c != '/') return false;
 return IsForm(t.LeftTree()) && IsForm(t.RightTree());
}
// в)
int ArExpr(CharTree const&t)
{char c = t.RootTree();
 if (c >= '0' && c <= '9') return (int)c - (int)'0';
 switch(c)
 {case '+': return ArExpr(t.LeftTree()) + ArExpr(t.RightTree());
  case '-': return ArExpr(t.LeftTree()) - ArExpr(t.RightTree());
  case '*': return ArExpr(t.LeftTree()) * ArExpr(t.RightTree());
  case '/': return ArExpr(t.LeftTree()) / ArExpr(t.RightTree());
 }
 return 99999; // добавено е за коректност на кода
}
// г)
void print_tree(CharTree const&t)
{char c = t.RootTree();
 if (c >= '0' && c <= '9') cout << c;
 else
 {cout << '(';
  print_tree(t.LeftTree());
  cout << c;
```

```
    print_tree(t.RightTree());
    cout << ')';
   }
  }
  void main()
  {CharTree t;
   t.Create();
   if (IsForm(t))
   {print_tree(t);
    cout << endl << ArExpr(t) << endl;
   }
   else cout << "Is not a formula.\n";
  }
```

**Задача 168.** Нека в двоичното дърво от тип char е записана формула според синтаксиса от предишната задача, но в качество на терминали се използват не само цифри, а и букви, играещи ролята на променливи. Да се напише функция, която:

а) опростява двоично дърво, представящо формула като заменя в него всички поддървета, съответстващи на формулите (f+0), (0+f), (f-0), (f*1), (1*f) и (f/1) с поддърво, съответстващо на формулата f, а поддърветата, съответстващи на формулите (f*0), (0*f) и (0/f) - с върха 0.

б) преобразува двоично дърво, представящо формула като заменя в него всички поддървета, съответстващи на формулите ((f1+f2)*f3), ((f1-f2)*f3), (f1*(f2+f3)), (f1*(f2-f3)), ((f1+f2)/f3) и ((f1-f2)/f3) - с поддървета, съответстващи на формулите ((f1*f3)+(f2*f3)), ((f1*f3)-(f2*f3)), ((f1*f2)+(f1*f3)), ((f1*f2)-(f1*f3)), ((f1/f3)+(f2/f3)) и ((f1/f3)-(f2/f3)) съответно.

Програма Zad168.cpp решава условие а) на задачата. Условие б) оставяме за самостоятелна работа.

```
// Program Zad168.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<char> CharTree;
```

```
void Reduce(CharTree &t)
{char c = t.RootTree();
 if (c == '+' || c== '-' || c == '*' || c ==' /')
 {CharTree t1, t2;
  t1 = t.LeftTree();
  t2 = t.RightTree();
  Reduce(t1);
  Reduce(t2);
  t.Create3(c,t1,t2);  // много важно!
  if (c == '+' && t1.RootTree() == '0' ||
      c == '*' && t1.RootTree() == '1')
  t = t2;
  else
  if ((c == '*' || c == '/') && t2.RootTree() == '1' ||
      (c == '+' || c == '-') && t2.RootTree() == '0')
  t = t1;
  else
  if (c == '*' && (t1.RootTree() == '0' ||
                   t2.RootTree() == '0') ||
      c == ’/’ &&  t1.RootTree() == ’0’)
  {CharTree t3;
   t.Create3('0', t3, t3);
  }
 }
}
void main()
{CharTree t;
 t.Create();
 Reduce(t);
 t.print();
}
```

**Задача 169.** Да се напише шаблон на булева функция, който реализира проверка за принадлежност на елемент в двоично дърво.

```
template <class T>
```

```
bool member(T a, tree<T> const&t)
{if (t.empty()) return false;
 if (a == t.RootTree()) return true;
 return member(a, t.LeftTree()) || member(a, t.RightTree());
}
```

Това решение е неефективно, тъй като ако елементът не се съдържа в дървото се налага обхождане на дървото с пълно изчерпване. Операциите включване и изключване на връх не се реализират добре за обикновените дървета. По-удобно за редица цели е използването на т. нар. **двоично наредени дървета** или **двоични справочници**.

## 16.2 Двоично наредено дърво

Предполагаме, че за елементите от тип T е установена наредба.

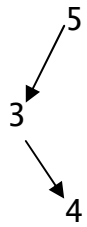
**Дефиниция:** Празното двоично дърво е двоично наредено дърво. Непразно двоично дърво, върховете на лявото поддърво на което са по-малки от корена, върховете на дясното поддърво са по-големи от корена и лявото, и дясното поддърво са двоично наредени дървета, се нарича **двоично наредено дърво от тип T**.

Нека t е двоично наредено дърво от тип T. *Включването на елемента a от тип T в t* се осъществява по следния начин:

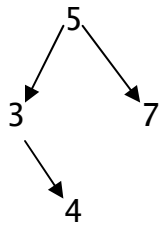
- ако t е празното двоично дърво, новото двоично наредено дърво е с корен елемента a и празни ляво и дясно поддървета.
- ако t не е празно и a е по-малко от корена му, елементът a се включва в лявото поддърво на t,
- ако t не е празно и a е не по-малко от корена му, елементът a се включва в дясното поддърво на t.

**Пример:** Ако в двоично нареденото дърво с корен 5 и празни поддървета се включи 3, ще се получи:

Ако към полученото двоично наредено дърво се включи 4, ще се получи:



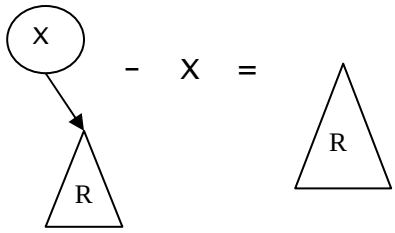

Ако към полученото двоично наредено дърво се включи 7, ще се получи:

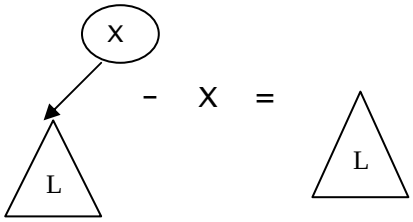


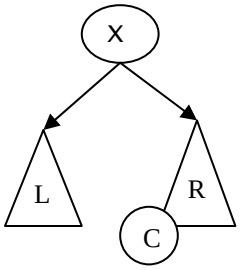

Включваният елемент "се спуска" по двоичното дърво, започвайки от корена и се насочва към лявото или дясното поддърво в зависимост от стойността си, докато намери празно място, което да заеме. Този начин на включване на елемент в двоично наредено дърво ще използваме за създаване на такива дървета.

Двоично наредената дърво притежава следното свойство: Обхождането на такова дърво по метода ЛКД сортира във възходящ ред елементите от върховете на дървото, а обхождането му по метода ДКЛ сортира в низходящ ред елементите от върховете на дървото.

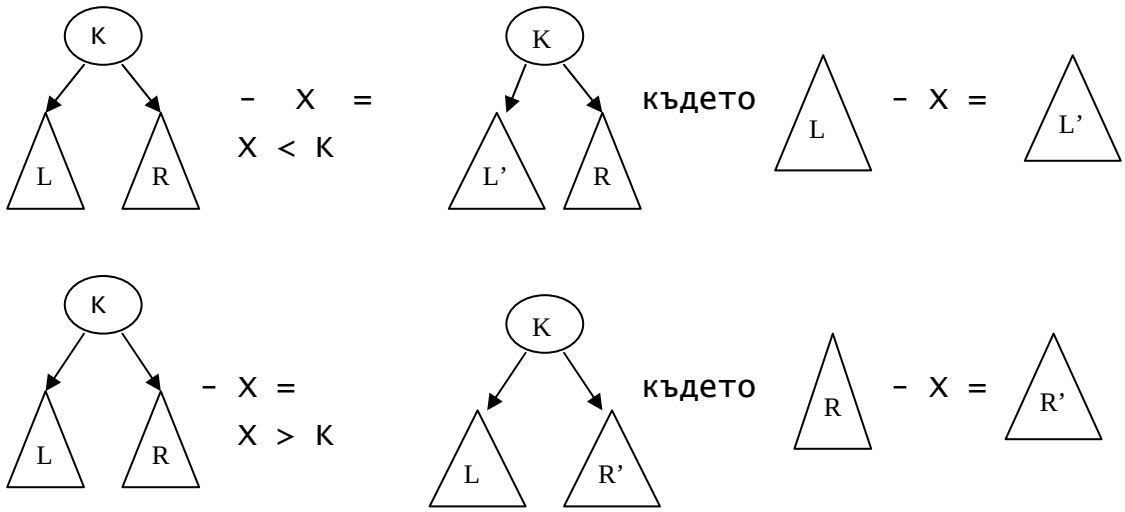
Изтриването на елемент от двоично наредено дърво се осъществява по следната схема:




където C е минималмия елемент на R, а R’ е R без минималмия му елемент.




Шаблонът на класа BinOrdTree реализира двоично наредено дърво от тип T.

```
template <class T>
struct node
{T inf;
 node *Left;
 node *Right;
};
template <class T>
class BinOrdTree
{public:
 BinOrdTree();
 ~BinOrdTree();
 BinOrdTree(BinOrdTree const&);
 BinOrdTree& operator=(BinOrdTree const&);
 bool empty()const;
 T RootTree() const;
 BinOrdTree LeftTree() const;
 BinOrdTree RightTree() const;
 void print() const
 {pr(root);
  cout << endl;
 }
```

```
 void AddNode(T const & x)
 {Add(root, x);
 }
 void DeleteNode(T const&);
 void Create3(T, BinOrdTree, BinOrdTree);
 void Create();
 void MinTree(T &, BinOrdTree &)const;
 private:
  node<T> *root;
  void DeleteTree(node<T>* &) const;
  void Copy(node<T> * &, node<T>* const&) const;
  void CopyTree(BinOrdTree const&);
  void pr(const node<T> *) const;
  void Add(node<T> *&, T const &) const;
 };
```

Голямата четворка е реализирана по същия начин като при обикновените двоични дървета.

```
template <class T>
BinOrdTree<T>::BinOrdTree()
{root = NULL;
}
template <class T>
BinOrdTree<T>::~BinOrdTree()
{DeleteTree(root);
}
template <class T>
BinOrdTree<T>::BinOrdTree(BinOrdTree<T> const& r)
{CopyTree(r);
}
template <class T>
BinOrdTree<T>& BinOrdTree<T>::operator=(BinOrdTree<T> const& r)
{if (this != &r)
 {DeleteTree(root);
  CopyTree(r);
 }
 return *this;
```

```
}
template <class T>
void BinOrdTree<T>::DeleteTree(node<T>* &p) const
{if (p)
 {DeleteTree(p->Left);
  DeleteTree(p->Right);
  delete p;
  p = NULL;
 }
}
template <class T>
void BinOrdTree<T>::CopyTree(BinOrdTree<T> const& r)
{Copy(root, r.root);
}
template <class T>
void BinOrdTree<T>::Copy(node<T> * &pos, node<T>* const &r) const
{pos = NULL;
 if (r)
 {pos = new node<T>;
  pos->inf = r->inf;
  Copy(pos->Left, r->Left);
  Copy(pos->Right, r->Right);
 }
}
```

Член-функциите empty, RootTree, LeftTree, RightTree, pr и print също са като тези на обикновените двоични дървета.

```
template <class T>
bool BinOrdTree<T>::empty()const
{return root == NULL;
}
template <class T>
T BinOrdTree<T>::RootTree()const
{return root->inf;
}
template <class T>
BinOrdTree<T> BinOrdTree<T>::LeftTree()const
```

```
{BinOrdTree<T> t;
 Copy(t.root, root->Left );
 return t;
}
template <class T>
BinOrdTree<T> BinOrdTree<T>::RightTree()const
{BinOrdTree<T> t;
 Copy(t.root, root->Right);
 return t;
}
template <class T>
void BinOrdTree<T>::pr(const node<T>*p) const
{if (p)
 {pr(p->Left);
  cout << p->inf << " ";
  pr(p->Right);
 }
}
```

Член-функцията Add е помощна. Използва се за реализиране на член-функцията AddNode на шаблона, чрез която се включва елемент в двоично наредено дърво по описания по-горе начин.

```
template <class T>
void BinOrdTree<T>::Add(node<T>* &p, T const & x)const
{if (!p)
 {p = new node<T>;
  p->inf = x;
  p->Left = NULL;
  p->Right = NULL;
 }
 else
 if (x < p->inf) Add(p->Left, x);
 else Add(p->Right, x);
}
```

Създаването на непразно двоично наредено дърво се осъществява чрез член-функцията Create. Тя реализира въвеждане на елементи и включването им чрез AddNode в двоично наредено дърво.

```
template <class T>
void BinOrdTree<T>::Create()
{root = NULL;
 T x; char c;
 do
 {cout << "> ";
  cin >> x;
  AddNode(x);
  cout << "next elem  y/n? "; cin >> c;
  } while (c == 'y');
}
```

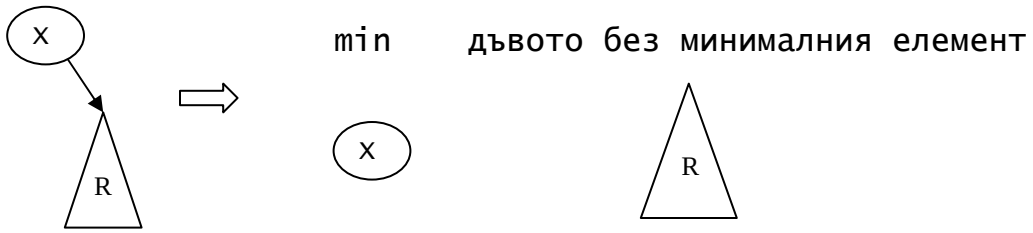
Член-функцията Create3 е същата като тази при ненаредените двоични дървета.

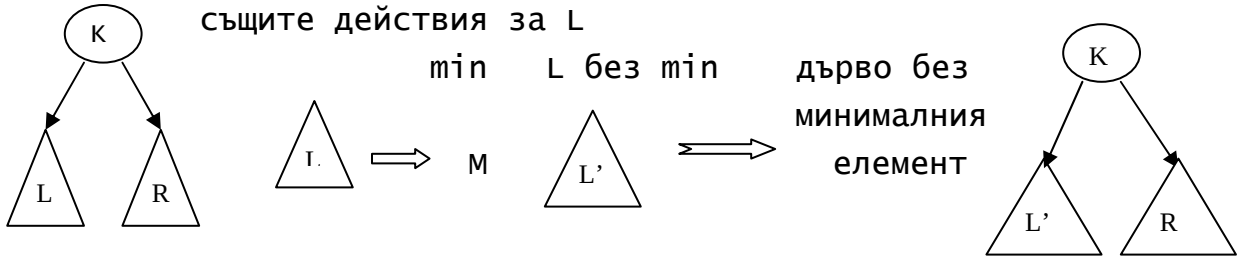
```
template <class T>
void BinOrdTree<T>::Create3(T x, BinOrdTree<T> l, BinOrdTree<T> r)
{root = new node<T>;
 root->inf = x;
 Copy(root->Left, l.root);
 Copy(root->Right, r.root);
}
```

Член-функцията MinTree намира минималния елемент на подразбиращото се (соченото от указателя this) двоично наредено дърво и подразбиращото се двоично наредено дърво без минималния му елемент. Реализира следния алгоритъм.

- Ако лявото поддърво на подразбиращото се двоично нареденото дърво е празно, минималният му елемент е корена, а дървото без минималния елемент е дясното му поддърво.
- Ако лявото му поддърво е непразно, на това двоично наредено дърво се намира минималния елемент и дървото без минималния елемент. След това се конструира двоично наредено дърво от корена, лявото поддърво без минималния му елемент и дясното поддърво на подразбиращото се дърво.

Тези действия могат да се опишат графично така:




```
template <class T>
void BinOrdTree<T>::MinTree(T &x, BinOrdTree<T> &mint) const
{T a = RootTree();
 if (LeftTree().empty())
 {x = a;
  mint = RightTree();
 }
 else
 {BinOrdTree<T> t1;
  LeftTree().MinTree(x, t1);
  mint.Create3(a, t1, RightTree());
 }
}
```

Член-функцията DeleteNode изтрива връх на подразбиращото се двоично наредено дърво. Реализира описания по-горе алгоритъм. Изключването трябва да се предшества от проверка дали изключваният елемент принадлежи на двоично нареденото дърво.

```
template <class T>
void BinOrdTree<T>::DeleteNode(T const& x)
{if (root)
 {T a = RootTree();
  BinOrdTree<T> t;
  if (a == x && LeftTree().empty()) *this = RightTree();
```

```
      else
      if (a == x && RightTree().empty()) *this = LeftTree();
      else
      if (a == x)
      {T c;
       RightTree().MinTree(c, t);
       Create3(c, LeftTree(), t);
      }
      else
      if (x < a)
      {t = *this;
       *this = LeftTree();
       DeleteNode(x);
       Create3(a, *this, t.RightTree());
      }
      else
      if (x > a)
      {t = *this;
       *this = RightTree();
       DeleteNode(x);
       Create3(a, t.LeftTree(), *this);
      }
     }
    }
```

Записваме този шаблон във файла BinOrdTree.cpp и ще го експериментираме с няколко задачи.

## Задачи върху двоично наредено дърво

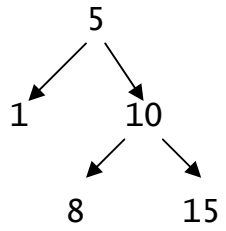
**Задача 170.** Да се напише програма, която въвежда редица от цели числа, сортира ги във възходящ ред, след което изключва указани върхове като запазва наредеността на елементите.

```
// Program Zad170.cpp
#include <iostream.h>
```

```
#include "BinOrdTree.cpp"
typedef BinOrdTree<int> IntTree;
void main()
{IntTree t;
 t.Create();
 t.print();
 int x;
 char c;
 do
 {cout << "x: ";
  cin >> x;
  t.DeleteNode(x);
  t.print();
  cout << "next: y/n: ";
   cin >> c;
 } while (c == 'y');
 t.print();
}
```

Експериментите с този и предходния шаблони извеждат в линеен вид двоичното дърво. За задача 170 този вид е подходящ, но има случаи, при които това не е така. Ще добавим към шаблона на класа BinOrdTree капсолираната процедура за извеждане pr1 и интерфейсната print_tree, които извеждат дървото, но завъртяно на 90 градуса по посока обратна на часовниковата стрелка.

**Пример:** Двоично-нареденото дърво




ще бъде изведено по следния начин:

```
        15
    10
        8
  5
    1
```

```
template <class T>
void BinOrdTree<T>::pr1(const node<T>* p, int n) const
{if (p)
 {n = n + 5;
  pr1(p->Right, n);
  cout << setw(n) << p->inf << endl;
  pr1(p->Left, n);
 }
}
```

и

```
tempalate <class T>
void BinOrdTree<T>::print_tree()const
 {pr1(root, 0);
  cout << endl;
 }
```

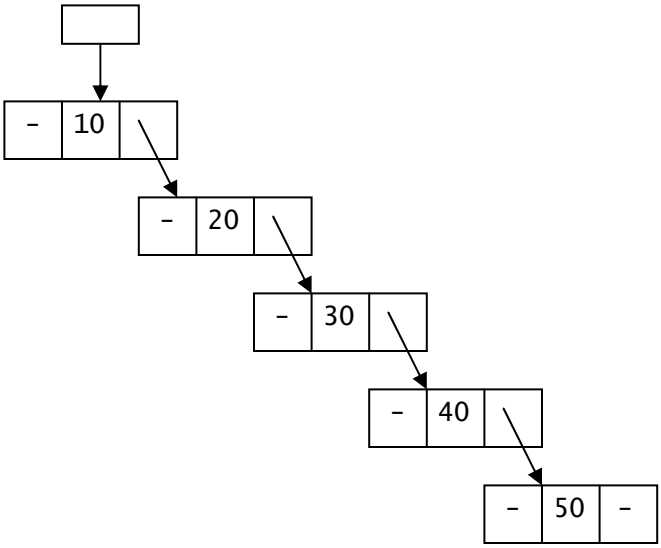
**Забележка:** Заради използването на модификатора setw се налага включването на заглавния файл iomanip.h.

**Задача 171.** Да се напише шаблон на булева функция, чрез който се проверява дали елемент се съдържа в двоично наредено дърво.

```
template <class T>
bool member(T a, BinOrdTree<T> const& t)
{if (t.empty()) return false;
 if (a == t.RootTree()) return true;
 if (a < t.RootTree()) return member(a, t.LeftTree());
 else return member(a, t.RightTree());
}
```

## 16.3. Балансирани и идеално балансирани двоично наредени дървета

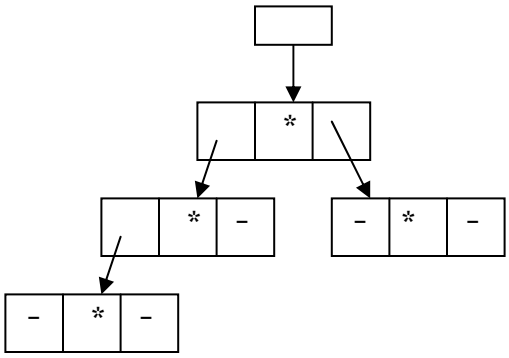
Ако елементите, които се включват в двоично наредено дърво са наредени в нарастващ ред, се получава двоично наредено дърво с линейна структура. Например, ако към празното двоично наредено дърво включим елементите: 10, 20, 30, 40, 50, се получава:



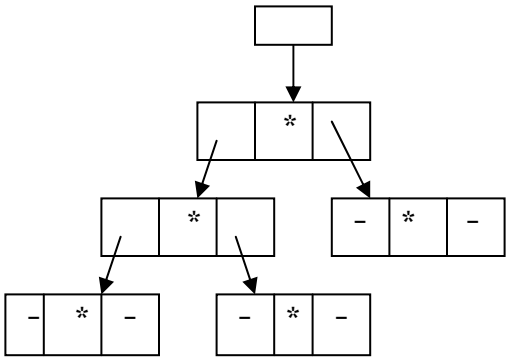

Търсенето на елемент в такова “изродено” дърво не е по-ефективно от това в свързаните списъци. Двоични дървета от този вид се наричат **силно небалансирани**.

**Дефиниция.** Двоично дърво се нарича **идеално (перфектно) балансирано** ако всеки негов връх има ляво и дясно поддърво, в които броят на възлите се различава най-много с 1.

**Примери:** Двоично дърво със следната структура




е идеално балансирано, но двоично дърво със структура:



не е идеално балансирано, тъй като лявото му поддърво има три върха, а дясното му – 1 (разликата е по-голяма от 1). Това двоично дърво е двоично дърво с балансирана височина.

**Дефиниция.** Двоично дърво се нарича **двоично дърво с балансирана височина** или по-просто **балансирано дърво**, ако за всеки негов връх височините (дълбочините) на лявото и дясното му поддървета се различават най-много с 1. Този вид двоични дървета се наричат също **AVL-дървета**.

От дефинициите и примера се вижда, че всяко идеално балансирано двоично дърво е дърво с балансирана височина, но обратното не е вярно.

Обикновено се изисква тези два вида двоични дървета да бъдат и наредени. Обаче алгоритъмът за включване на елемент в двоично наредено дърво, който използвахме за създаване на двоично наредено дърво, не създава идеално балансирано двоично наредено дърво, нито балансирано двоично наредено дърво. Операцията за изключване на елемент също не запазва идеалната балансираност или само балансираността на дървото.

Има ефективен алгоритъм за създаване на балансирани двоично наредени дървета, който при включване и изключване на елемент запазва балансираността на двоично нареденото дърво. За идеално балансираното двоично наредено дърво такъв не съществува.

Съществува прост алгоритъм за създаване на идеално балансирано двоично наредено дърво при следните ограничения:

- елементите, които ще се включват към празното двоично наредено дърво се подават в нарастващ ред;
- предварително е известен броят на върховете на дървото.

Към шаблона на класа BinOrdtree ще добавим и конструктор, който създава идеално балансирано двоично наредено дърво с n елемента.

```
template <class T>
BinOrdTree<T>::BinOrdTree(int n)
{if (n == 0) root = NULL;
 else
 {int nLeft = (n-1)/2,
```

```
        nRight = n – nLeft - 1;
    BinOrdTree<T> t1(nLeft);
    T x; cin >> x;
    BinOrdTree<T> t2(nRight);
    Create3(x, t1, t2);
   }
  }
```

Като използва n, алгоритъмът намира теглата nLeft и nRight на поддърветата на дървото, където

n = nLeft + nRight + 1 и |nLeft-nRight| ≤ 1.

Създава тройната кутия и я запълва в последователността: конструиране на лявото поддърво с nLeft върха, въвеждане на корена и конструиране на дясното поддърво с nRight върха. Конструирането на лявото и дясното поддърво се осъществява чрез рекурсивно обръщение към конструктора на идеално балансирано двоично наредено дърво.

## 16.4 Граф

### 16.4.1 Дефиниране на граф

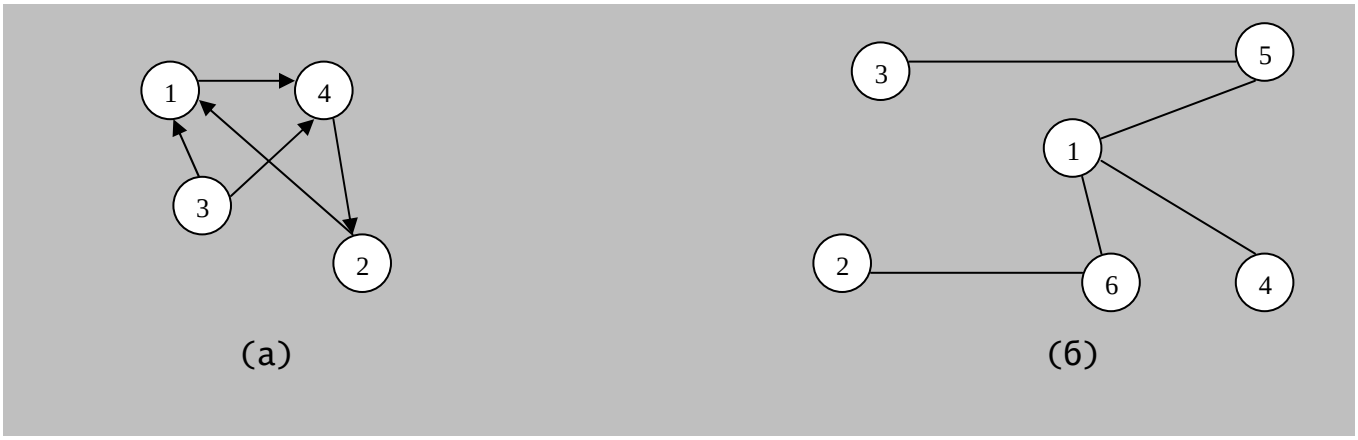
Структурата от данни граф ни е вече доста позната, тъй като свързаните списъци и двоичните дървета са нейни частни случаи. Освен това в Глави 10 и 13 на книгата Увод в програмирането на базата на езика C++, с която предполачм, че читателят е запознат, реализирахме последователното представяне на граф и предложихме алгоритми за търсене на път в граф.

**Логическо описание**

Графът се определя като множество от върхове заедно с множество от ребра. Всяко ребро се задава чрез двойка върхове. Ако ребрата са зададени като наредени двойки <i, j> от върхове, се наричат **дъги**. Такъв граф се нарича **ориентиран**. Ако ребрата на графа са зададени само чрез ненаредени двойки от върхове, графът се нарича **неориентиран**.

Ребрата (дъгите) могат да се свързват с етикети от произволен вид (имена, числа), в зависимост от конкретната задача. Тези етикети се наричат още **тегла**.

**Примери:** На Фиг. 16.2 (а) е даден ориентиран граф с цели числа във върховете, а на Фиг. 16.2 (б) – неориентиран граф.




Фиг. 16.2 Примери за ориентиран и неориентиран граф

Теорията на графите е интересен раздел на математиката. На базата на нея са реализирани редица полезни приложения. Ще разгледаме някои приложения на тази теория.

Ще използваме само ориентирани графи. Неориентираните графи ще представяме като ориентирани.

**физическо представяне**

Съществуват два основни начина за представяне на граф:

- последователно;
- свързано.

*Последователното представяне* се реализира чрез така наречената **матрица на съседство**. За граф с n върха, номерирани с 1, 2, ..., n, матрицата на съседство има n реда и n стълба. Ако <i, j> е дъга ориентирана от връх i до връх j в графа, елементът в i-тия ред и j-тия стълб на матрицата на съседство е равен на 1, в противен случай той е 0.

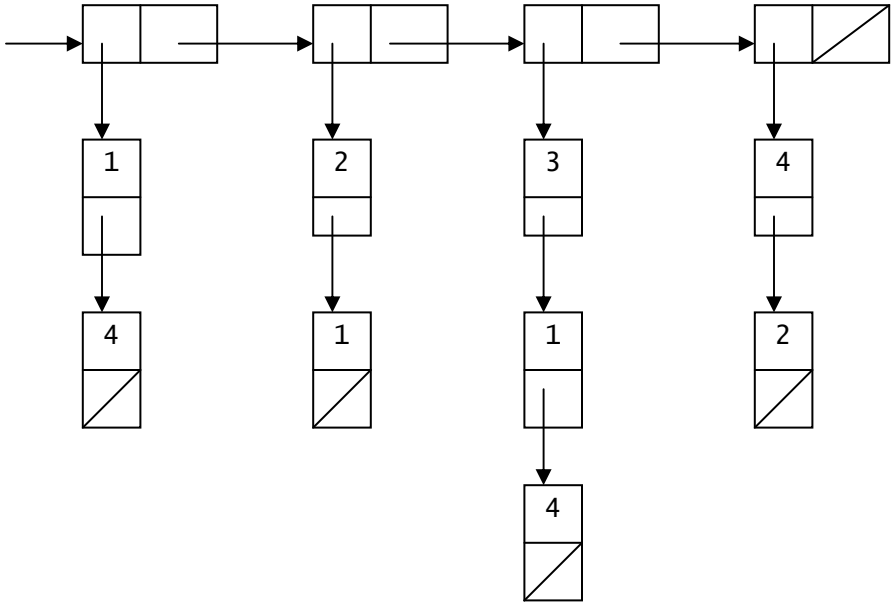
**Пример:** Матрицата на съседство за графа от Фиг 16.2 (а) има вида:

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Това представяне на практика не се използва често, тъй като при големи графи матриците на съседство заемат прекалено много памет и при много алгоритми водят до квадратично време за изпълнение. Матрици на съседство, с голям брой нулеви елементи, се наричат **разредни**.

*Свързаното представяне* се базира на свързаните списъци. Има различни реализации. Сравнително просто и удобно е да се използва свързан списък от толкова елемента, колкото са върховете на графа. Елементите на списъка са списъци като всеки подсписък да започва с връх i на графа и да съдържа всички върхове j, така че има дъга от i до j.

**Пример:** Това представяне за графа от Фиг. 16.2 (а) има вида:




Това представяне ще реализираме чрез шаблона на класа LList, дефиниращ свързан списък с една връзка и записан във файла L-List.cpp. Специализациите

```
typedef LList<int> IntList;
typedef LList<IntList> IntGraph;
```

определят класа IntGraph, задаващ граф с цели числа във върховете и реализиращ предложеното по-горе представяне.

**Задача 172.** Да се създаде непразен граф с цели числа във върховете, след което да се изведе.

За създаването на графа ще реализираме следния алгоритъм. Докато желаем, въвеждаме цяло число, означаващо връх на графа, след което го включваме като връх. След това отново докато желаем въвеждаме наредени двойки от върхове, означаващи дъги в графа и ги включване. За реализиране на тези действия ще дефинираме следните помощни процедури:

- void AddTop(int a, IntGraph &g), включваща върха a в графа g;
- void AddRib(int a, int b, IntGraph &g), включвща ребро от връх a до връх b на графа g.

За реализиране на включването на ребро, а също и за други цели, е полезна функцията

elem<int>* Point(int a, IntGraph &g)

която намира указател към двойната кутия, в информационната част на която е записан върхът a на графа g.

```
// Program Zad172.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<int> IntList;
typedef LList<IntList> IntGraph;
elem<int>* Point(int a, IntGraph &g)
{g.IterStart();
 elem<IntList>*p;
 elem<int> *q;
 do
 {p = g.Iter();
  p->inf.IterStart();
  q = p->inf.Iter();
 } while(q->inf != a);
 return q;
```

```
}
void AddTop(int a, IntGraph &g)
{IntList l;
 l.ToEnd(a);
 g.ToEnd(l);
}
void AddRib(int a, int b, IntGraph &g)
{elem<int> * q = Point(a, g), *p;
 while (q->link) q = q->link;
 p = new elem<int>;
 p->inf = b;
 p->link = NULL;
 q->link = p;
}
void create_graph(IntGraph &g)
{char c;
 do
 {cout << "top_of_graph: ";
  int x; cin >> x;
  AddTop(x, g);
  cout << "Top y/n? "; cin >> c;
 } while (c == 'y');
 cout << "Ribs:\n";
 do
 {cout << "start top: ";
  int x; cin >> x;
  cout << "end top: ";
  int y; cin >> y;
  AddRib(x, y, g);
  cout << "next: y/n? "; cin >> c;
 } while (c == 'y');
}
void LList<IntList>::print()
{elem<IntList> *p = Start;
 while (p)
 {p->inf.print();
```

```
    p = p->link;
   }
   cout << endl;
  }
  void main()
  {IntGraph g;
   create_graph(g);
   g.print();
  }
```

Друга реализация на create_graph, която в някои случаи е по-удобна е:

```
  void create_graph(IntGraph &g)
  {cout << "Number of tops: ";
   int n; cin >> n;
   for (int i = 1; i <= n; i++)
   {cout << "top_of_graph: ";
    int x; cin >> x;
    AddTop(x, g);
    cout << "Number of Tops from " << x <<  " To? ";
    int k; cin >> k;
    for (int j = 1; j <= k; j++)
    {cout << "top: ";
     int y; cin >> y;
     AddRib(x, y, g);
    }
   }
  }
```

**Задача 173.** Да се напише булева функция, която установява дали съществува ацикличен път от един до друг връх на граф.

Обикновено се реализира следният алгоритъм. Съществува път от връх a до връх b, ако:

- a==b или
- съществува връх c, така че от a до c има ребро и има път от връх c до връх b.

Ако a не съвпада с b и е намерен връх c, така че от a до c има ребро, възможно е задачата за търсене дали има път от c до b да се сведе до търсене на път от a до b и да се стигне до зациклене. За да се избегне зациклянето може да се използва помощен списък l, в който да се записват върховете, които са преминати в процеса на търсене на път. За следващ връх от пътя да се избира само такъв, който не се съдържа в списъка l. Отначало l ще бъде празния списък, а ако път съществува, в l ще се намери един път от връх a до връх b на графа g. Булевата функция way реализира търсене с връщане назад, разгледан подробно в глава 13.

```
bool way(int a, int b, IntGraph &g, IntList &l)
{l.ToEnd(a);
 if (a == b) return true;
 elem<int> * q = Point(a, g);
 q = q->link;
 while (q)
 {if (!member(q->inf, l) && way(q->inf, b, g, l)) return true;
  DeleteLast(l);  // връщане назад
  q = q->link;
 }
 return false;
}
```

Функцията way използва вече известната функция

```
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if (!p) return false;
 int y;
 l.DeleteElem(p, y);
 return x == y || member(x, l);
}
```

а също и процедурата DeleteLast, която изтрива последния елемент на списък.

```
void DeleteLast(IntList &l)
{l.IterStart(); int x;
 elem<int>*p = l.Iter();
```

```
 while (p->link) p = l.Iter();
 l.DeleteElem(p, x);
}
```

**Задача 174.** Да се напише булева функция, която намира всички ациклични пътища от един до друг връх на граф.

Процедурата

```
void allways(int a, int b, IntGraph &g, IntList &l)
```

намира всички пътища от връх a до връх b на графа g. Всеки път се конструира в списъка l, след което се извежда. Глобалната булева променлива flag получава стойност true ако съществува път от a до b в g. Инициализирана е с false. Процедурата allways реализира търсене с връщане назад.

```
bool flag = false;
void allways(int a, int b, IntGraph &g, IntList &l)
{l.ToEnd(a);
 if (a == b)
 {flag = true;
  l.print();
 }
 else
 {elem<int> * q = Point(a, g);
  q = q->link;
  while (q)
  {if (!member(q->inf, l))
   {allways(q->inf, b, g, l);
    DeleteLast(l); // връщане назад
   }
   q = q->link;
  }
 }
}
```

## Задачи

**Задача 1.** Да се дефинира процедура, която извежда отначало всички положителни, а след това всички отрицателни върхове на двоично дърво от тип int.

**Задача 2.** Дадено е двоично дърво от тип int. Да се напише функция, която определя има ли сред върховете на двоичното дърво поне два върха с равни стойности.

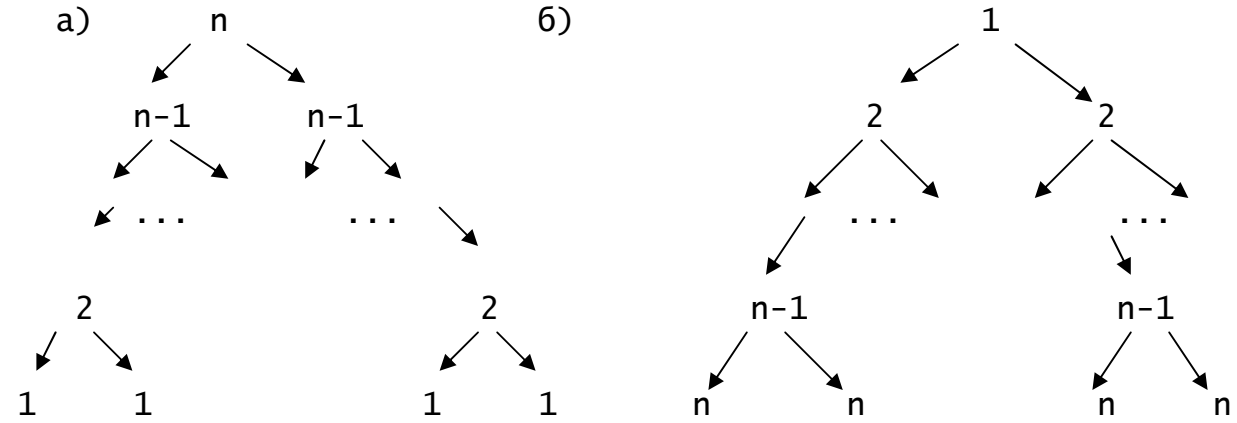
**Задача 3.** Дадено е двоично дърво от тип int. Да се напише функция, която намира сумата на четните елементите от върховете на двоичното дърво.

**Задача 4.** Да се напише функция или процедура, която:

а) определя броя на включванията на елемента a в двоичното дърво d;

б) намира броя на върховете на n-то ниво на непразното двоично дърво d (коренът се счита за връх от 0-во ниво).

**Задача 5.** Да се напише процедура CreateBinTree(d, n), където n е положително цяло число, която създава следното двоично дърво:




**Задача 6.** Да се напише програма, която за даден аритметичен израз от вида:

<АИ> ::= <терминал>|
(<АИ> <знак> <АИ>);

<знак> ::= +|-|*|/;

<терминал> ::= 0|1|...|9,

намира и извежда правия полски запис, който съответства на израза.

**Задача 7.** Да се напише програма, която за даден аритметичен израз от вида:

```
<АИ> ::= <терминал>|
         (<АИ> <знак> <АИ>);
<знак> ::= +|-|*|/;
<терминал> ::= 0|1|...|9,
```

намира и извежда обратния полски запис, който съответства на израза.

**Задача 8.** Даден е свързан списък, съдържащ реални числа. Да се напише програма, която от елементите на списъка генерира двоично наредено дърво.

**Задача 9.** Даден е стек от реални числа. Да се напише програма, която сортира елементите на стека като за целта използва двоично наредено дърво.

**Задача 10.** Да се напише програма, която установява дали съществува път между два върха на дадено двоично дърво. Ако съществува път, да се намери пътя, а също и дължината му.

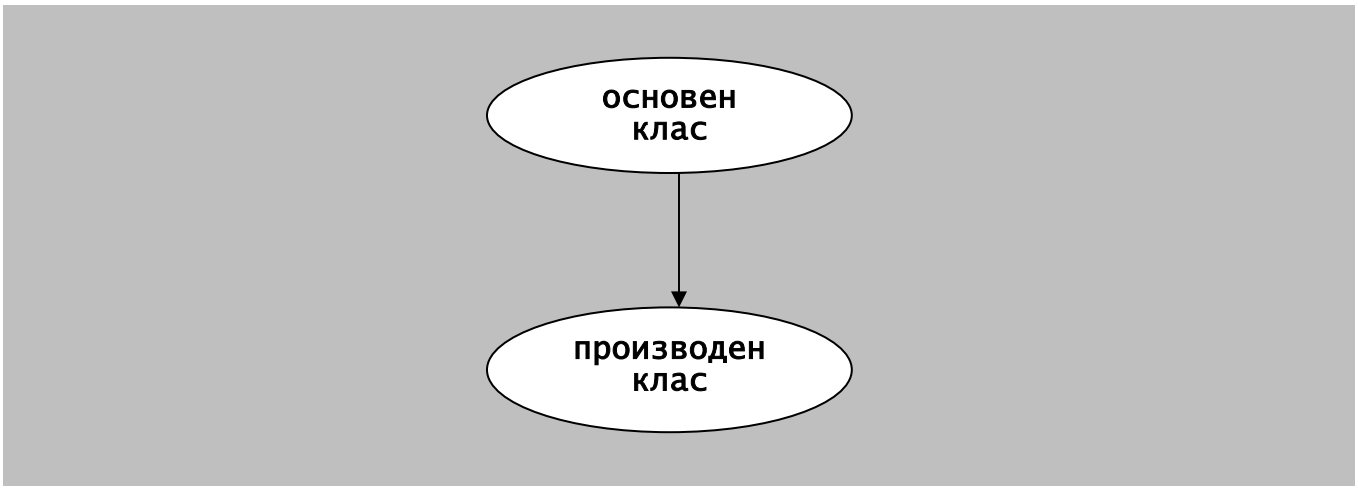
## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. А. Берстисс, Структуры данных, Москва, Статистика, 1974.
3. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
4. Л. Амерал, Алгоритми и структури от данни в C++, София, ИК СОФТЕХ, 2001.
5. М. Тодорова, Програмиране на Паскал, София, Полипринт, 1993.

# 17

# Наследяване. Производни класове

Производните класове и наследяването са една от най-важните характеристики на обектно-ориентираното програмиране (ООП). Чрез механизма на наследяване от съществуващ клас се създава нов клас. Класът от който се създава се нарича **базов (основен) клас**, а този, който е създаден - **производен**. На фиг. 17.1 е показана най-проста връзка между основен и производен клас.




Фиг. 17.1 Най-проста връзка между основен и производен клас

Понятията основен и производен клас са относителни, тъй като производен клас може да е основен за други класове, а основен – да е производен от други основни класове. Производният клас може да наследи компонентите на един или няколко базови класа. В първия случай наследяването се нарича **единично (просто)**, а във втория – **множествено**.

Дефинирането на производни класове е еквивалентно на конструирането на йерархии от класове. *Защо се налага дефинирането на производни класове? В кои случаи и как се прави това? Какви са предимствата от дефинирането на производни класове?* На тези въпроси ще дадем отговор в следващите разглеждания.

Ако множество от класове имат общи данни и методи, тези общи части могат да се обособят като основни класове, а всяка от останалите части да се дефинира като производен клас на съответния основен клас. Така се прави икономия на памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти.

При конструирането на производни класове е достатъчно да се разполага само с обектните модули на основните класове, а не с техния програмен код. Това позволява да бъдат създавани библиотеки от класове, които да бъдат използвани при създаването на производни класове.

Тези предимства, а също възможността за реализиране на полиморфизъм, мотивират въвеждането на производни класове.

## 17.1 Дефиниране на производни класове

Подобно на обикновените, производните класове се дефинират като се *декларира класът* и се *дефинират неговите методи*. Синтаксисът на декларацията е даден на Фиг. 17.2.

```
Декларираме на производен клас
<декларация_на_производен_клас> ::=
class <име_на-производен_клас> :
       [<атрибут_за_област>]опц <име_на_базов_клас>
       {, [<атрибут_за_област>]опц <име_на_базов_клас>}опц
{<декларация_на_компоненти>
};
<име_на-производен_клас> ::= <идентификатор>
<атрибут_за_област> ::= public | private | protected
<име_на_базов_клас> ::= <идентификатор>
```

Фиг. 17.2 Декларираме на производен клас

Пред всяко име на базов клас *може* да се постави запазената дума public, private или protected. Нарича се **атрибут за област**, тъй като определя областта на наследените членове. Употребата на атрибутите за област е различна от тази за обявяване на секции в тялото на класа. Ако атрибут за област е пропуснат, подразбира се private. Атрибутът protected е включен в новите версии на езика и не се използва много често.

**Примери:**

1. Декларацията:

```
class der : base1, base2, base3
{...
};
```

определя производен клас der с три основни класа base1, base2 и base3. Тъй като атрибутът за област е пропуснат и за трите базови класа, подразбира се private, т.е. декларацията е еквивалентна на

```
class der : private base1, private base2, private base3
{...
};
```

2. Декларацията:

```
class der : public base1, base2, base3
{...
};
```

е еквивалентна на

```
class der : public base1, private base2, private base3
{...
};
```

3. Декларацията:

```
class der : protected base1, base2, public base3
{...
};
```

е еквивалентна на

```
class der : protected base1, private base2, public base3
{...
};
```

*Декларациите на компонентите на производен клас, а също дефинициите на неговите методи не се различават от съответните при обикновените класове.*

Множеството от компонентите на един производен клас се състои от компонентите на неговите базови класове и компонентите, декларирани в самия производен клас. Оттук произлиза и терминът наследяване. Механизмът, чрез който производният клас получава компонентите на базовия, се нарича наследяване. Когато производният клас има няколко базови класа, той наследява компонентите на всеки от тях. Наследяването в този случай е множествено.

*Процесът на наследяване се изразява в следното:*

- наследяват се данните и методите на основния клас;
- получава се достъп до някои от наследените членове на основния клас;
- производният клас "познава" реализацията само на основния клас, от който произлиза;
- производният клас може да е основен за други класове.

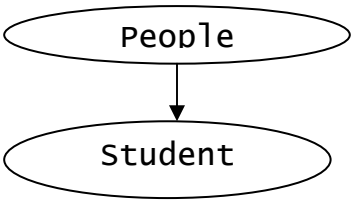
*Производният клас може да дефинира допълнително:*

- свои член-данни;
- методи, аналогични на тези на основния клас, а също и нови.

Дефинираните в производния клас данни и методи се наричат **собствени**. Чрез примери ще илюстрираме дефинирането на производни класове с единично наследяване.

**Задача 157.** Да се напише програма, която дефинира клас People, определящ човек по име и единен граждански номер (ЕГН), а също производен клас Student на класа People, който определя понятието студент като човек, който има факултетен номер и среден успех. Да се дефинира обект от клас Student и се изведе дефинираният обект.

Програма Zad157.cpp решава задачата. Тя реализира йерархията:



```
// Program Zad157.cpp
#include <iostream.h>
#include <string.h>
// дефиниция на базовия клас People
class People
{public:
  void ReadPeople(char *, char *);
  void PrintPeople() const;
 private:
  char * name;
  char * egn;
};
void People::ReadPeople(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
// дефиниция на производния клас Student
class Student : People
{public:
  void ReadStudent(char *, char *, long, double);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
void Student::ReadStudent(char *str, char * num,
                          long facn, double u)
{ReadPeople(str, num);
 facnom = facn;
```

```
 usp = u;
}
void Student::PrintStudent() const
{PrintPeople();
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
}
int main()
{Student stud;
 stud.ReadStudent("Ivan Ivanov", "8206123422", 42444, 6.0);
 stud.PrintStudent();
 return 0;
}
```

Чрез класа People е представено понятието човек, характеризиращо човек с име и ЕГН, реализирани чрез член-данните name и egn от тип char*. Капсулирани са чрез декларирането им като private. Освен член-данни класът съдържа и методите ReadPeople и PrintPeople, образуващи интерфейса на класа (обявени са като public). Чрез ReadPeople се инициализират обектите на класа People, а чрез PrintPeople се извеждат върху екрана стойностите на член-данните name и egn. Ще напомним, че заделената от методите динамична памет не се освобождава автоматично при унищожаване на обектите. Освобождаването на тази памет трябва да стане явно, чрез оператора delete. В тази част умишлено методът ReadPeople не е реализиран като конструктор, не е дефиниран също и деструктор. Това е заради някои особености при дефинирането на тези методи, които особености ще разгледаме по-късно в тази глава.

Класът Student, дефиниран в програмата, представя понятието студент, като реализира следното определение: Студент, това е човек, който има факултетен номер и се характеризира със среден успех. Това дава основание Student да бъде определен като производен клас на класа People. В резултат, класът Student има осем компоненти. Четири от тях (name, egn, ReadPeople и PrintPeople) са наследени от базовия клас People и четири (facnom, usp, ReadStudent и PrintStudent) са декларирани в него. Тъй като не е указан атрибут за област на базовия

клас People, подразбира се private. В този случай производният клас Student наследява всички член-данни и член-функции на основния клас като private. Освен това той получава възможността да използва всички компоненти на основния клас, които не са private (в случая ReadPeople и PrintPeople). Така член-функциите ReadStudent и PrintStudent нямат пряк достъп до наследените член-данни name и egn. Затова достъпът е реализиран чрез методите ReadPeople и PrintPeople.

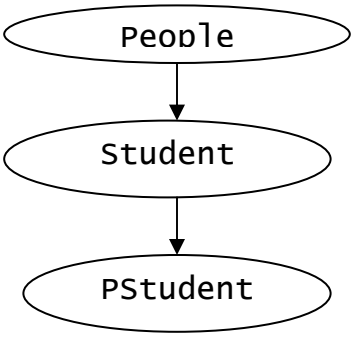
Производният клас е дефиниран след като вече е дефиниран базовият клас, от който той произлиза. Чрез него се разширява декларацията на съществуващ клас. Разширяемостта на класовете е една от важните характеристики на ООП.

Чрез следващата задача ще покажем възможността производен клас да е основен за друг клас, т.е. да бъде създадена верига от наследени класове.

**Задача 158.** Да се дефинира клас PStudent, производен на класа Student, реализиращ понятието студент от платена форма на обучение.

```
class PStudent : public Student
{public:
  void ReadPStudent(char *, char *, long, double, double);
  void PrintPStudent() const;
 private:
  double tax; // такса за обучението на студента
};
void PStudent::ReadPStudent(char *str, char *num, long facn,
                            double u, double t)
   {ReadStudent(str, num, facn, u);
    tax = t;
   }
   void PStudent::PrintPStudent() const
   {PrintStudent();
    cout << “Tax: “ << tax << endl;
   }
```

Връзката между класовете People, Student и PStudent е следната:




Класът PStudent е производен на класа Student с атрибут за област public. В този случай PStudent наследява всички компоненти на класа Student (собствени и наследени от People) като запазва вида им, т.е. собствените методи ReadStudent и PrintStudent продължават да са public, а собствените член-данни facnom и usp и всички наследени от People продължават да са private в класа PStudent. Това е така, тъй като атрибутът за област на класа Student е private, заради което всички компоненти на People са наследени от Student като private и отново като private се наследяват и от класа PStudent.

От тези примери се вижда, че на атрибута за област е отредена важна роля. Семантиката му ще разгледаме в следващата част.

## 17.2 Наследяване и достъп до наследените компоненти

Ще напомним, че в рамките на един клас (без наследяване), protected частта има аналогична роля като тази на private частта. До компоненти от тип protected имат пряк достъп само член-функции и приятелски функции на класа.

Атрибутът за област на базовия клас в декларацията на производния клас (public, private или protected) управлява механизма на наследяване и определя какъв да бъде режимът на достъп до наследените членове. Таблицата от Фиг. 17.3 показва наследяванията на компоненти на основен клас в зависимост от атрибута за област.

| Атрибут за област | Компонента на основен клас, определена като | Наследява се като |
|---|---|---|
| public | private<br>public<br>protected | private<br>public<br>protected |
| private | private<br>public<br>protected | private<br>private<br>private |
| protected | private<br>public<br>protected | private<br>protected<br>protected |

Фиг. 17.3 Наследявания на компоненти на основен клас в производен

От таблицата се вижда, че:

· Ако базовият клас е декларирен като public в производния клас, всички private, public и protected компоненти на базовия клас се наследяват съответно като private, public и protected компоненти на производния клас.

**Пример:** Ако

```
class base                  class der1 : public base
{private: int b1;           {private: int d1;
 protected: int b2;          protected: int d2;
 public: int b3();           public: int d3();
};                          };
```

можем да си мислим, че der1 е клас от вида:

```
class der1
{private:
  int b1;
  int d1;
 protected:
  int b2;
  int d2;
 public:
  int b3();
```

```
    int d3();
  };
```

· Ако базовият клас е декларира като private в производния клас, всички негови компоненти се наследяват като private.

**Пример:** Ако

```
class base                  class der2 : private base
{private: int b1;           {private: int d1;
 protected: int b2;          protected: int d2;
 public: int b3();           public: int d3();
};                          };
```

можем да си мислим, че der2 е клас от вида:

```
class der2
{private:
  int b1;
  int b2;
  int b3();
  int d1;
 protected:
  int d2;
 public:
  int d3();
};
```

· Ако базовият клас е декларира като protected в производния клас, private компонентите му се наследяват като private, а public и protected – като protected.

**Пример:** Ако

```
class base                  class der3 : protected base
{private: int b1;           {private: int d1;
 protected: int b2;          protected: int d2;
 public: int b3();           public: int d3();
};                          };
```

можем да си мислим, че der3 е клас от вида:

```
class der3
{private:
```

```
    int b1;
    int d1;
   protected:
    int b2;
    int d2;
    int b3();
   public:
    int d3();
  };
```

*Наследените компоненти обаче се различават от декларираните в производния клас по правата за достъп. Производният клас има пряк достъп до компонентите, декларирани като public и protected, но няма пряк достъп до декларираните като private в базовия клас.* Достъпът до private компонентите на базовия клас се извършва чрез неговия интерфейс.

Таблицата от Фиг. 17.4 показва прекия достъп на член-функции на производния клас (ПД) и външния достъп на производния клас (ВД) до компонентите на базовия клас.

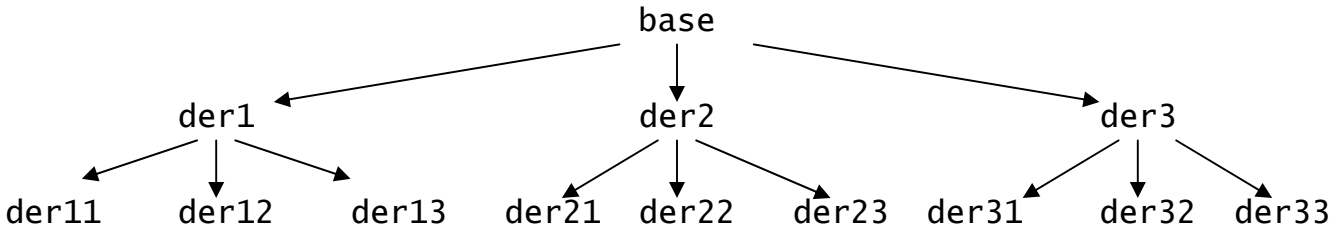
| компонента на базов клас | производен клас с атрибут public | | производен клас с атрибут private | | производен клас с атрибут protected | |
|---|---|---|---|---|---|---|
| | ПД | ВД | ПД | ВД | ПД | ВД |
| public | да | да | да | не | да | не |
| protected | да | не | да | не | да | не |
| private | не | не | не | не | не | не |

Фиг. 17.4 Достъп на член-функции и на обекти на производния клас до компонентите на базовия клас

Да се върнем към означенията от последните три примера. Собствените компоненти на класа der1 са видими навсякъде в класа. Те имат пряк достъп до компонентите b2 и b3() на base, но нямат пряк достъп до private-компонентата b1 на base. Същото се отнася и за класовете der2 и der3. Освен това, обект от клас der1 има пряк достъп public-компонентите b3() – наследена и d3() – собствена за der1; обект от клас der2 има пряк достъп единствено до собствената

public-компонента d3(), тъй като всички наследени от base компоненти се наследяват като private и обект от клас der3 има също пряк достъп единствено до собствената public-компонента d3(), тъй като public и protected компонентите на base се наследяват като protected в der3.

С цел разясняване на процеса на наследяване, ще разгледаме и следната йерархия на класове:




с декларации:

```
class base
{private: int a1;
 protected: int a2;
 public: int a3();
} b;
class der1: public base  class der2: private base  class der3:protected base
{private: int a4;         {private: int a7;        {private: int a10;
 protected: int a5;        protected: int a8;       protected: int a11;
 public: int a6();         public: int a9();        public: int a12();
} d1;                     } d2;                    } d3;
class der11:public der1  class der12:private der1 class der13:protected der1
{private: int a13;        {private: int a16;       {private: int a19;
 protected: int a14;       protected: int a17;      protected: int a20;
 public: int a15();        public: int a18();       public: int a21();
} d4;                     } d5;                    } d6;

class der21:public der2  class der22:private der2 class der23:protected der2
{private: int a22;        {private: int a25;       {private: int a28;
 protected: int a23;       protected: int a26;      protected: int a29;
 public: int a24();        public: int a27();       public: int a30();
} d7;                     } d8;                    } d9;

class der31:public der3  class der32:private der3 class der33:protected der3
```

```
{private: int a31;        {private: int a34;       {private: int a37;
 protected: int a32;       protected: int a35;      protected: int a38;
 public: int a33();        public: int a36();       public: int a39();
} d10;                     } d11                    } d12;
```

Всеки обект на класовете der1, der2 и der3 притежава 6 компоненти – 3 собствени и 3 наследени от базовия клас base, а всеки от класовете deri1, deri2 и deri3 притежава 9 компоненти: 3 собствени, 6 наследени от клас deri (i = 1, 2, 3).

Ще разгледаме достъпа на компонентите, дефинирани в класовете derij (i, j = 1, 2 3) до компонентите на класовете, предшестващи ги в йерархията.

Член-функциите на класа der1i (i = 1, 2, 3) имат пряк достъп до (могат да използват) всички собствени компоненти, до наследените като protected и public a5 и a6() на класа der1 и до наследените като protected и public a2 и a3() на класа base. Достъпът им до a1 и a4 се осъществява чрез интерфейса на съответните базови класове.

Член-функциите на класа der2i (i = 1, 2, 3) имат пряк достъп до (могат да използват) всички собствени компоненти и до наследените като protected и public a8 и a9() на класа der2. Тъй като базовият на класа der2 клас base е с атрибут за област private, компонентите на base се наследяват от der2 като private. Това е причината те да не са достъпни за член-функциите на класовете der2i (i = 1, 2, 3).

Член-функциите на класа der3i (i = 1, 2, 3) имат пряк достъп до (могат да използват) всички собствени компоненти, до наследените като protected и public собствени компоненти a11 и a12() на класа der3 и до наследените като protected a2 и a3() на класа base. Достъпът им до останалите компоненти се осъществява чрез интерфейса на съответните базови класове.

Външният достъп до компонентите на класовете ще определим чрез достъпа на дефинираните обекти на класовете от горната йерархия.

| обект | възможност за достъп до компонента: |
|---|---|
| b | a3() |
| d1 | a6(), a3() |

| | |
|---|---|
| d2 | a9() |
| d3 | a12() |
| d4 | a15(), a6(), a3() |
| d5 | a18() |
| d6 | a21() |
| d7 | a9(), a24() |
| d8 | a27() |
| d9 | a30() |
| d10 | a12(), a33() |
| d11 | a36() |
| d12 | a39() |

Ще се спрем на някои от често срещаните случаи за достъп до членове на производен и основен клас, а също на достъпа на външни функции до наследен компонент. Ще изкажем и някои правила за достъп до компоненти на базови и производни класове, които ще подкрепим с още примери.

**· Достъп до членове на основен клас чрез дефиниции на методи на производен клас**

**В сила са следните правила за достъп:**

*– Методите на производен клас (без значение на атрибута за област) нямат директен достъп до членовете от private-секцията на основния му клас*

**Примери:**

а) Класът Student е производен на класа People. Атрибутът за област не е указан явно, заради което се подразбира private. Методите на Student нямат пряк достъп до private членовете name и egn на People.

б) Класът PStudent е производен на Student. Атрибутът за област е public. От Фиг. 17.3 следва, че видът на наследените секции на Student се запазва. Методите на PStudent нямат пряк достъп както до собствените private компоненти facnom и usp на Student, така и до наследените от People private членовете name и egn.

Ще отбележим също,че тъй като атрибутът за област на класа People в Student е private, всички компоненти на People са private в Student и са недостъпни пряко в PStudent.

*- В дефинициите на собствени методи на производния клас могат да се използват методите от секциите public и protected на основния му клас*

**Примери:**

а) Тъй като методите на Student нямат пряк достъп до name и egn, инициализацията на тези компоненти в ReadStudent става чрез метода ReadPeople на класа People, който е обявен в public секцията на People.

б) Тъй като методите на PStudent нямат пряк достъп до facnom, usp, name и egn, инициализацията им в ReadPStudent става чрез метода ReadStudent на класа Student, който е обявен в public секцията на Student.

*- В дефинициите на собствени методи на производния клас може директно да се използват член-данните на секцията protected на основния му клас*

Ще илюстрираме това правило като извършим промени в програма Zad157.cpp.

**Задача 159.** Да се промени програма Zad157.cpp така, че освен класовете People и Student да включва и наследения от Student клас PStudent. Освен това, методите на производните класове да могат пряко да използват наследените член-данни на основните им класове.

```
// Program Zad159.cpp
#include <iostream.h>
#include <string.h>
class People
{public:
  void ReadPeople(char *, char *);
  void PrintPeople() const;
 protected: // вместо private:
```

```
  char * name;
  char * egn;
};
void People::ReadPeople(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
class Student : public People // вместо private:
{public:
  void ReadStudent(char *, char *, long, double);
  void PrintStudent() const;
 protected: // вместо private
  long facnom;
  double usp;
};
void Student::ReadStudent(char *str, char * num,
                          long facn, double u)
{name = new char[strlen(str)+1]; // използваме член-данните
 strcpy(name, str);              // name и egn
 egn = new char[11];
 strcpy(egn, num);
 facnom = facn;
 usp = u;
}
void Student::PrintStudent() const
{cout << "Ime: " << name << endl;   // използваме член-данните
 cout << "EGN: " << egn << endl;    // name и egn
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
}
```

```
class PStudent : public Student
{public:
  void ReadPStudent(char *, char *, long, double, double);
  void PrintPStudent() const;
 protected:
  double tax;
};
void PStudent::ReadPStudent(char *str, char *num, long facn,
                            double u, double t)
{name = new char[strlen(str)+1]; // пряк достъп до
 strcpy(name, str);              // name, egn, facnom и usp
 egn = new char[11];
 strcpy(egn, num);
 facnom = facn;
 usp = u;
 tax = t;
}
void PStudent::PrintPStudent() const
{cout << "Ime: " << name << endl;      // пряк достъп до
 cout << "EGN: " << egn << endl;       // name, egn, facnom и usp
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
 cout << "Tax: " << tax << endl;
}
int main()
{PStudent stud;
 stud.ReadPStudent("Ivan Ivanov", "8206123422", 42444, 6.0, 1000);
 stud.PrintPStudent();
 return 0;
}
```

Методите ReadStudent и PrintStudent на дефинирания в тази програма производен клас Student имат пряк достъп до член-данните name и egn на своя базов клас People, тъй като последните са декларирани като protected. Аналогично, методите ReadPStudent и PrintPStudent на дефинирания също в тази програма производен клас PStudent имат пряк достъп до собствените член-данни facnom и usp на базовия си клас

Student, тъй като последните са декларирани като protected и до name и egn на базовия клас People за класа Student, тъй като те са декларирани в People също като protected, но атрибутът за област на People в Student е public, заради което се наследяват от Student като protected. Функцията main е външна както за класа Student, така и за PStudent и няма пряк достъп до техните protected компоненти. Затова компилаторът ще сигнализира грешка при опит за пряк достъп до компонентите name, egn, facnom и usp в main.

· **Достъп до методи чрез обекти на основния и производния клас**

Обект на основен клас има пряк достъп до всички свои компоненти, обявени като public и няма пряк достъп до компонентите, обявени като private и protected. Обект на производен клас има пряк достъп до public компонентите на собствения си и компонентите на основния клас, наследени в производния клас като public. От фиг. 17.3 се вижда, че последното е възможно, ако атрибутът за област на производния клас е public и компонентата е в public секция на основния клас.

Нека сме в дефинициите на Zad159.cpp и дефинираме следните обекти:

```
People pe;
Student stud;
PStudent pstud;
```

Ще напомним, че Student е производен клас на основен клас People с атрибут за област public, а PStudent е производен клас на основния клас Student също с атрибут за област public. Обектът pe има пряк достъп до public методите на People, stud има пряк достъп до public методите на People и Student, а pstud има пряк достъп до public методите на People, Student и PStudent, т.е. допустими са обръщенията:

```
pe.ReadPeople("Ivan Ivanov", "5804134986");
pe.PrintPeople();
stud.ReadStudent("Pavel Dimov", "4806193046", 30100, 4.50);
stud.ReadPeople("Pavel Dimov", "4806193046");
stud.PrintPeople();
stud.PrintStudent();
pstud.ReadPStudent("Pavel Dimov","4806193046", 30100, 4.50, 500);
```

```
pstud.ReadStudent("Pavel Dimov", "4806193046", 30100, 4.50);
pstud.ReadPeople("Pavel Dimov", "4806193046");
pstud.PrintPeople();
pstud.PrintStudent();
pstud.PrintPStudent();
```

Да се върнем към програма Zad157.cpp. В нея класът People е основен на класа Student с атрибут за област private. Student наследява всички секции на People като private. Следователно обектите на Student нямат пряк достъп до методите на People. Ако имаме дефинициите:

```
People pe;
Student stud;
```

допустими са обръщенията

```
pe.ReadPeople("Ivan Ivanov", "5804134986");
pe.PrintPeople();
stud.ReadStudent("Pavel Dimov", "4806193046", 30100, 4.50);
stud.PrintStudent();
```

а обръщенията:

```
stud.ReadPeople("Ivan Ivanov", "5804134986");
stud.PrintPeople();
```

са недопустими.

**· Достъп на основен клас до членове на производен клас и обратно**

Методите на основния клас нямат достъп до членове на производен клас. Причината е, че когато основният клас се дефинира, не е ясно какви производни класове ще произхождат от него.

Производният клас също няма привилигировън достъп до членове на основния клас. От фиг. 17.3 се вижда, че производните класове нямат достъп до методите, обявени като private в основния клас.

Допустими са редица присвоявания между обекти на основния и производния клас. Ще ги разгледаме подробно в следващи части на главата. Засега ще отбележим само, че за реализирането им се извършват редица преобразувания.

· **Достъп на функции приятели на производен клас до компоненти на основния му клас**

Ще напомним, че функциите-приятели на клас не са елементи на класа, на който са приятели. Те са външни функции, получили привилигировън достъп до компонентите на класа.

Функциите приятели на производен клас имат същите права на достъп като член-функциите на производния клас. *Имат пряк достъп до всички компоненти, декларирани в класа и до public и protected компонентите на основния клас. Декларацията за приятелство не се наследява. Функция приятел на базовия клас не е приятел (освен ако не е декларирана като такава) на производния клас.*

**Пример:** В резултат от изпълнението на програмата:

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 void readbase(int x, int y)
 {a1 = x;
  a2 = y;
 }
 void a3() const
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: int d1;
 protected: int d2;
 public:
 friend void f(der &d, int x, int y, int z)
 {cout << "friend function f(): " << endl;
  d.d1 = x;
  d.d2 = y;
  d.a2 = z;
  cout << "d.a3(): " << endl;
```

```
      d.a3();
      cout << "d.d3(): " << endl;
      d.d3();
     }
     void readder(int x, int y, int z, int t)
     {readbase(x, y);
      d1 = z;
      d2 = t;
     }
    void d3() const
    {cout << "d1: " << d1 << endl
          << "d2: " << d2 << endl
          << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
     }
    };
    void main()
    {der x;
     x.readder(10, 20, 30, 40);
     x.d3();
     f(x, 100, 200, 300);
    }
```

се получава:

```
    d1: 30
    d2: 40
    a2: 20
    a3():
    a1: 10
    a2: 20
    friend function f()
    d.a3()
    a1: 10
    a2: 300
    d.d3():
    d1: 100
```

```
d2: 200
a2: 300
a3():
a1: 10
a2: 300
```

Ще отбележим, че ако der е основен клас за класа der1, функцията приятел f, дефинирана в примера по-горе, е видима в класа der1, но f не е функция приятел за der1.

**Забележки:**

1. Използването на секция protected позволява пряк достъп на производния клас до нейните компоненти. По такъв начин се нарушава принципът на капсулиране на данните, но пък дадената "привилегия" повишава ефективността на генерирания код.

2. Дефинирането на производни класове с атрибут за област private предизвиква забрана на достъпа на обект на класа до интерфейса на базовия му клас. Това се прави когато не трябва да се използва интерфейса на базовия клас, а се налага да се преработи и всички негови полезни функции да бъдат предефинирани.

## 17.3 Предефиниране на компоненти

Базовият и производният клас могат да притежават собствени компоненти с еднакви имена. В този случай производният клас ще притежава компоненти с еднакви имена. Обръщението към такава компонента чрез обект от производния клас извиква декларираната в класа компонента, т.е. *името на собствената компонента е с по-висок приоритет от това на наследената*. За да се изпълни "покритата" наследена компонента се указва пълното й име, т.е.

```
<име_на_клас>::<компонента>
```

където <име_на_клас> е името на основния клас.

**Пример:** В резултат от изпълнението на програмата

```
#include <iostream.h>
class base
{public:
 void init(int x)
```

```
    {bx = x;
    }
    void display() const
    {cout << " class base: bx= " << bx << endl;
    }
    protected:
     int bx;
    private:
   // …
   };
   class der: public base
   {public:
    void init(int x)
    {bx = x;
     base::bx = x + 5;
    }
    void display() const
    {cout << " class der: bx = " << bx;
     cout << " base::bx = " << base::bx << endl;
    }
    protected:
     int bx;
    private:
    //...
   };
   void main()
   {base b;
    der d;
    b.init(5); d.init(10);
    b.display(); d.display();
    d.base::init(20);
    d.base::display();
    d.display();
    b.display();
   }
```

се получава:

```
class base: bx = 5
class der: bx = 10 base::bx = 15
class base: bx = 20
class der: bx = 10 base::bx = 20
class base: bx = 5
```

## 17.4 Конструктори, операторни функции за присвояване и деструктори на производни класове

Обикновените конструктори, конструкторът за присвояване, операторната функция за присвояване и деструкторът са методи, за които не важат правилата за достъп при наследяване. ***Тези методи (с някои изключения) не се наследяват от производния клас.*** Ако например конструктор можеше да бъде наследен, той щеше да инициализира само наследената част. Нормално е конструкторът на производен клас да инициализира както наследената, така и собствената част на класа. Същото се отнася и за деструкторът. Това е причината, заради която конструкторите и деструкторът на основния клас не се наследяват от производния клас. Възможно е обаче конструкторът на производния клас да активира конструктор на основния клас, който пък да инициализира наследената част. Производният клас не наследява и създадените от програмиста конструктор за присвояване и предефинирания оператор за присвояване на обекти =.

Следователно, голямата четворка на основния клас не се наследява от производния клас. Това е следствие на особената роля на четворката.

Дефинирането и използването на голямата четворка за производния клас ще разгледаме на няколко стъпки. За да разграничим конструктора за присвояване от останалите конструктори, последните ще наричаме обикновени или само конструктори.

### 17.4.1 Обикновени конструктори и деструктор

**Конструктори**

Обикновените конструктори на базовия и на производния клас изпълняват инициализиращи функции. Принципен е въпросът *как и от кого да се реализира инициализирането на наследената част на производния клас.* Най-естествено е това да се направи от конструкторите на производния клас. Но ако това е така, конструкторите на производния клас трябва да имат достъп до наследените, при това най-често, private компоненти на основния клас. Това е в противоречие на принципа за капсулиране на информацията. Затова инициализирането на собствените и наследените членове е разделено между конструкторите на производния и основния клас.

Конструкторите на производния клас инициализират само собствените член-данни на класа. Наследените член-данни на производния клас се инициализират от конструктор на основния клас. Това се осъществява като в дефиницията на конструктора на производния клас се укаже обръщение към съответен конструктор на основния клас (Фиг. 17.4).

**Дефиниция на конструктор на производен клас**

**Синтаксис**

```
<дефиниция_на_конструктор_на_производен_клас> ::=
<име_на_производен_клас>::<име_на_производен_клас>(<параметри>)
                          <инициализиращ_списък>
{<тяло>
}
<инициализиращ_списък> ::= <празно>|
                           : <име_на_основен_клас>(<параметри_i>)
                           {,<име_на_основен_клас>(<параметри_i>)}_опц
<параметри> ::= <празно>|
                <параметър>|
                <параметри>, <параметър>
<параметър> ::= <тип> <име_на_параметър>
<име_на_параметър> ::= <идентификатор>
```

<параметри$_i$> е конструкция, която има синтаксиса на <фактически_параметри> от дефиницията на обръщение към функция (глава 8), а <тяло> се определя като тялото на който и да е конструктор.

Фиг. 17.4 Дефиниция на конструктор на производен клас

При единично наследяване в инициализиращия списък на производния клас е указано не повече от едно обръщение към конструктор на основен клас. При множествено наследяване в инициализиращия списък може да са указани няколко обръщения към конструктори на основни класове. За разделител се използва символът запетая.

**Забележка.** Обръщенията към конструкторите на основни класове се обявяват в дефиницията на конструктора на производния клас, а не в неговата декларация в тялото на производния клас.

В тази глава ще останем в означенията на единичното наследяване.

Ще отбележим също, че <параметри$_i$> са изрази или идентификатори, съответстващи по брой, тип и смисъл на формалните параметри на съответния конструктор на базовия клас, т.е. обръщението

<име_на_основен_клас>(<параметри$_i$>)

трябва да се оформи според дефиницията на конструктора на основния клас. Имената на параметри от конструктора на производния клас могат да се използват за фактически параметри в обръщението към конструктора на основния клас.

**Пример:** Да разгледаме следните изкуствени класове:

```
// дефиниция на базовия клас base
class base
{private: int a1;
 protected: int a2;
 public:
 base()  // конструктор по подразбиране
 {a1 = 0;
  a2 = 0;
 }
 base(int x)   // конструктор с един параметър
 {a1 = x;
 }
 base(int x, int y)  // конструктор с два параметъра
 {a1 = x;
  a2 = y;
 }
 void a3()
```

```
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
// дефиниция на производния клас der
class der : public base
{private: int d1;
 protected: int d2;
 public:
 der(int x, int y, int z, int t) : base(x, y) // конструктор
 {d1 = z;
  d2 = t;
 }
 void d3()
 {cout << "d1: " << d1 << endl
       << "d2: " << d2 << endl
       << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
  }
 };
```

Тъй като в инициализиращия списък на конструктора на класа der участва двуаргументният конструктор на base, наследените компоненти се инициализират от него. В резултат от изпълнението на фрагмента:

```
der x(1, 2, 3, 4);
x.d3();
```

се получава:

```
d1: 3
d2: 4
a2: 2
a3():
a1: 1
a2: 2
```

Ако вместо двуаргументния конструктор на основния клас дефиницията на конструктора на производния клас der използва подразбиращият се конструктор на base, т.е.

```
der(int x, int y, int z, int t) : base()
 {d1 = z;
  d2 = t;
 }
```

наследените компоненти от базовия клас base ще се инициализират от подразбиращия се за base конструктор. В резултат от изпълнението на фрагмента:

```
der x(1, 2, 3, 4);
x.d3();
```

се получава:

```
d1: 3
d2: 4
a2: 0
a3():
a1: 0
a2: 0
```

Тази дефиниция на конструктора на der е еквивалентна на дефиницията:

```
der(int z, int t)
 {d1 = z;
  d2 = t;
 }
```

т.е. подразбиращият се конструктор на основния клас може да бъде пропуснат в инициализиращия списък на конструктора на производния клас.

Ако вместо двуаргументния конструктор на основния клас дефиницията на конструктора на производния клас der използва едноаргументния конструктор на base, т.е.

```
der(int x, int y, int z, int t) : base(x)
 {d1 = z;
  d2 = t;
 }
```

наследената компонента a1 от базовия клас base ще се инициализира с x, а компонентата a2 ще остане неинициализирана и в резултат от изпълнението на фрагмента:

```
der x(1, 2, 3, 4);
x.d3();
```

ще се получи:

```
d1: 3
d2: 4
a2: -858993460
a3():
a1: 1
a2: -858993460
```

Към примера ще отбележим изрично, че в инициализиращия списък може да участва не повече от едно обръщение към конструктор на един и същ базов клас, т.е. дефиниция от вида:

```
der(int x, int y, int z, int t) : base(), base(x, y)
 {d1 = z;
  d2 = t;
 }
```

е недопустима.

*Семантика*

Дефинирането на обект от производен клас предизвиква създаване на “неявен” обект от базовия му клас и добавяне на декларираните в производния клас компоненти. Това означава, че ако и базовият, и основният клас имат конструктори, то първо се извиква конструкторът на базовия, а след това – конструкторът на производния клас.

В случая на множествено наследяване, извикването на конструктор на производен клас води до извикването на указаните в дефиницията му конструктори на неговите основни класове и след завършване на тяхното изпълнение се изпълнява <тяло> на конструктора на производния клас. Процедурата е следната:

- заменят се формалните с фактическите параметри във всяко обръщение към конструктор на основен клас, след което се изпълнява обръщението;
- изпълняват се операторите в тялото на конструктора на производния клас.

Ако производният клас има член-данни, които са обекти, техните конструктори се извикват след изпълнението на обръщенията към конструкторите на основните класове от инициализиращия списък и преди изпълнението на операторите в тялото на конструктора на производния

клас. Конструкторите на обектите се извикват по реда на тяхното декларираме в тялото на производния клас.

**Пример:** Резултатът от изпълнението на програмата

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 base()
 {cout << "constructor base() \n";
  a1 = 0;
  a2 = 0;
 }
 base(int x, int y)
 {cout << "constructor base(" << x << "," << y << ")\n";
  a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: base d1; // не може base d1(1, 5). Защо?;
 protected: base d2;
 public:
 der(int x, int y) : base(x, y)
 {cout << "constructor der\n";
 }
 void d3()
 {d1.a3();
  d2.a3();
  cout << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
```

```
 }
};
void main()
{der x(1, 2);
 x.d3();
}
```

е следният:

```
constrictor base(1, 2)
constrictor base()
constrictor base()
constrictor der
a1: 0
a2: 0
a1: 0
a2: 0
a2: 2
a3():
a1: 1
a2: 2
```

Ще се отбележим специално на някои случаи:

· **В основния клас не е дефиниран конструктор**

В този случай в инициализиращият списък не се отбелязва нищо. Наследената част на производния клас остава неинициализирана.

**Пример:** Резултатът от изпълнението на програмата:

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 void readbase(int x = 0, int y = 0)
 {a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
```

```
          << "a2: " << a2 << endl;
    }
   };
   class der : public base
   {private: int d1;
    protected: int d2;
    public:
    der(int x, int y)
    {cout << "constructor der\n";
     d1 = x;
     d2 = y;
    }
    void d3()
    {cout << "d1: " << d1 << endl;
     cout << "d2: " << d2 << endl;
     cout << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
    }
   };
   void main()
   {der x(1, 2);
    x.d3();
   }
```

e:

```
   constructor der
   d1: 1
   d2: 2
   a2: -858993460
   a3():
   a1: -858993460
   a2: -858993460
```

отрицателните стойности на a1 и a2 показват, че тези данни са неинициализирани.

· **Основният клас има само един конструктор с параметри, който не е подразбиращия се**

Възможни са:

*а) в производния клас е дефиниран конструктор*

Тогава трябва да има задължително обръщение към него в инициализиращия списък. Изпълнява се по начина, описан по-горе.

**Пример:** Програмата

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 base(int x, int y)
 {a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: int d1;
 protected: int d2;
 public:
 der(int x, int y)      // няма обръщение към конструктор на base
 {cout << "constructor der\n";
  d1 = x;
  d2 = y;
 }
 void d3()
 {cout << "d1: " << d1 << endl;
  cout << "d2: " << d2 << endl;
  cout << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
```

```
    }
   };
   void main()
   {der x(1,2);
    x.d3();
   }
```

съобщава грешката:

```
primer.cpp(19):error C2512: 'base' : no appropriate default
                              constructor available
```

*б) в производния клас не е дефиниран конструктор*

В този случай компилаторът ще сигнализира за грешка. Необходимо е да се създаде конструктор за производния клас, който да активира конструктора на основния клас.

**Пример:** Програмата

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 base(int x, int y)
 {a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: int d1;
 protected: int d2;
 public:
 void d3()
 {cout << "d1: " << d1 << endl;
  cout << "d2: " << d2 << endl;
```

```
     cout << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
    }
   };
   void main()
   {der x;
    x.d3();
   }
```

издава следното съобщение за синтактична грешка:

```
primer.cpp(28):error C2512: 'der': no appropriate default
               constructor available
```

**· Основният клас има няколко конструктора в т. число подразбиращ се конструктор**

Възможни са:

*а) в производния клас е дефиниран конструктор*

Тогава може да не се посочва конструктор за основния клас в инициализиращия списък. Ако не е посочен, компилаторът се обръща към подразбиращия се конструктор на основния клас.

Пример вече беше даден.

*б) в производния клас не е дефиниран конструктор*

В този случай компилаторът автоматично създава подразбиращ се конструктор за производния клас. Последният активира и изпълнява подразбиращият се конструктор на основния клас. Собствените членове на подразбиращия се клас са инициализирани неопределено.

**Пример:** В резултат от изпълнението на програмата

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 base()
 {a1 = 0;
```

```
     a2 = 0;
    }
    void a3()
    {cout << "a1: " << a1 << endl
          << "a2: " << a2 << endl;
    }
   };
   class der : public base
   {private: int d1;
    protected: int d2;
    public:
    void d3()
    {cout << "d1: " << d1 << endl;
     cout << "d2: " << d2 << endl;
     cout << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
    }
   };
   void main()
   {der x;
    x.d3();
   }
```

се получава:

```
   d1: -858993460
   d2: -858993460
   a2: 0
   a3():
   a1: 0
   a2: 0
```

**Деструктори**

Деструкторите на един производен клас и на неговите основни класове се изпълняват в ред, обратен на реда на изпълнение на техните конструктори. Най-напред се изпълнява деструкторът на производния

клас, след това се изпълняват деструкторите на неговите основни класове.

**Пример:** Резултатът от изпълнението на програмата:

```
#include <iostream.h>
class A
{public:
 A(){cout << “Конструктор на клас A\n”;}
 ~A(){cout << “Деструктор на клас A\n”;}
};
class B : public A
{public:
 B(){cout << “Конструктор на клас B\n”;}
 ~B(){cout << “Деструктор на клас B\n”;}
};
class C : public B
{public:
 C(){cout << “Конструктор на клас C\n”;}
 ~C(){cout << “Деструктор на клас C\n”;}
};
void main()
{C x;
}
```

е:

```
Конструктор на класа A
Конструктор на класа B
Конструктор на класа C
Деструктор на клас C
Деструктор на клас B
Деструктор на клас A
```

При създаването на обекта x се извиква конструкторът на класа C. Тъй като C е производен на класа B и инициализиращият му списък е празен, се извиква конструкторът на класа B, който започва да създава “неявен” обект от клас B. Но класът B е производен на класа A и инициализиращият му списък е празен. Това предизвиква обръщение към подразбиращия се конструктор на класа A. Заради това отначало се изпълнява конструкторът на класа A, след това се довършва създаването

на обекта от клас B като се извиква конструкторът му. Най-накрая се извиква конструкторът на класа C за да завърши създаването на обекта x. При завършване изпълнението на тялото на функцията main започва процес на разрушаване на обекта x. Това предизвиква обръщение към деструктора на класа C, след това – към деструктора на класа B, след него – към деструктора на класа A и най-накрая обектът x се разрушава.

Тъй като член-данните на класа People от задача 157 са реализирани в областта за динамично разпределение на паметта, добре е за този клас да се реализира голямата четворка. Това ще направим стъпка по стъпка в следващите разглеждания. Засега ще променим класовете People, Student и PStudent като включим в тях конструктори и деструктори.

**Задача 160.** Да се дефинират повторно класовете People, Student и PStudent от задача 159, така че инициализиращите действия да се изпълняват от подходящи конструктори. Разрушителните действия да се извършват от деструктори.

Програма Zad160.cpp решава задачата.

```
// Program Zad160.cpp
#include <iostream.h>
#include <string.h>
// декларация на класа People
class People
{public:
  People(char * = "", char * = "");
  void PrintPeople() const;
  ~People();
 private:
  char * name;
  char * egn;
};
// дефиниция на конструктора на People
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
```

```
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
// дефиниция на метода PrintPeople
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
// дефиниция на деструктора на People
People::~People()
{cout << "~People(): " << endl;
 delete name;
 delete egn;
}
// декларация на класа Student
class Student : People
{public:
  Student(char * = “”, char * = “”, long = 0, double = 0);
  void PrintStudent() const;
  ~Student()
  {cout << "~Student(): " << endl;
  }
 private:
  long facnom;
  double usp;
};
//дефиниция на конструктора на класа Student
Student::Student(char *str, char * num, long facn,
  double u) : People(str, num)
{facnom = facn;
 usp = u;
}
// дефиниция на метода PrintStudent
void Student::PrintStudent() const
{PrintPeople();
```

```
 cout << "Fac. nomer: " << facnom << endl;
 cout << "Uspeh: " << usp << endl;
}
// декларация на класа PStudent
class PStudent : public Student
{public:
  PStudent(char * = “”, char * = “”, long = 0,
           double = 0, double = 0);
  ~PStudent()
  {cout << "~PStudent() \n";
  }
  void PrintPStudent() const;
 protected:
  double tax;
};
// дефиниция на конструктора на класа PStudent
PStudent::PStudent(char *str, char *num, long facn,
          double u, double t) : Student(str, num, facn, u)
{tax = t;
}
// дефиниция на метода PrintPStudent
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
{People pe;
 pe.PrintPeople();
 PStudent PStud("Ivan Ivanov", "8206123422", 42444, 6.0, 4567);
 PStud.PrintPStudent();
}
```

**Резултат:**

```
Ime:
EGN:
Ime: Ivan Ivanov
EGN: 8206123422
```

```
Fac.nomer: 42444
Uspeh: 6
Tax: 4567
~PStudent()
~Student()
~People()
~People()
```

Тази програма илюстрира използването на конструктори и деструктори на производни класове. Тя се различава от програма Zad159.cpp по това, че методите ReadPeople, ReadStudent и ReadPStudent на класовете People, Student и PStudent са заменени с конструктори. Освен това е добавен деструктор на класа People, който освобождава паметта на динамичните член-данни name и egn на People. Деструкторите на класовете Student и PStudent са напълно излишни, тъй като собствените им член-данни не са динамични. Дефинирани са с цел илюстрация на реда на изпълнението на деструкторите на класовете. При създаването на обекта PStud се извиква конструкторът на класа PStudent. Преди изпълнението на тялото му се прави обръщение към конструктора Student("Ivan Ivanov", "8206123422", 42444, 6.0), т.е. започва създаване на обект от класа Student. Преди изпълнението на тялото на този конструктор се изпълнява обръщението People("Ivan Ivanov", "8206123422"), което инициализира компонентите name и egn с "Ivan Ivanov" и "8206123422" съответно. Процесът на оценяване продължава с изпълнение на тялото на конструктора на класа Student, в резултат на което компонентите facnom и usp се инициализират с 42444 и 6 съответно. Най-накрая се изпълнява тялото на конструктора на класа PStudent, при което компонентата tax на обекта PStud се инициализира с 4567. Обръщението Stud.PrintPStudent() извежда отначало наследените член-данни, а след това и собствените член-данни на класа PStudent. Накрая се разрушава обектът PStud като преди това се извикват деструкторите на PStudent, Student и People.

Ако заменим дефинирания в класа People конструктор с два подразбиращи се параметъра с конструкторите

```
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
```

```
    strcpy(name, str);
    egn = new char[11];
    strcpy(egn, num);
   }
```

и

```
   People::People()
   {name = “”;
    egn = “”;
   }
```

дефиницията

```
   People pe;
```

ще предизвика обръщение към конструктора People() и член-данните на pe ще се инициализират с празния низ. При завършване на блока, изпълнението на деструктора ~People() ще направи опит да разруши несъществуващи връзки на name и egn с динамичната памет. Това ще предизвика грешка по време на изпълнение.

Ще отбележим също, че в инициализиращия списък на производния клас участват обръщения само към неговите базови класове. Дефиниция от вида:

```
   PStudent::PStudent(char *str, char *num, long facn,
             double u, double t) : People(str, num)
   {tax = t;
   }
```

предизвиква синтактична грешка – People не е основен клас на класа PStudent.

Ще отбележим също още една **често допускана грешка.**

В основния клас динамична променлива е дефинирана като protected. По такъв начин тя е видима и може пряко да се използва от всички член-функции на производния клас. Тъй като тази динамична променлива е наследена член-данна на производния клас, често се прави опит заетата от тази променлива памет да се освободи два пъти – веднъж от деструктора на базовия и веднъж от деструктора на производния клас. Това предизвиква грешка по време на изпълнение.

**Пример:** Дефиницията на деструктора на класа Student:

```
   class People
   {public:
```

```
  People(char * = “”, char * = “”);
  void PrintPeople() const;
  ~People();
 protected:
  char * name;
  char * egn;
};
People::People(char *str, char *num)
...
void People::PrintPeople() const
...
People::~People()
{cout << "~People(): " << endl;
 delete name;
 delete egn;
}
class Student : People
{public:
  Student(char * = “”, char * = “”, long = 0, double = 0);
  void PrintStudent() const;
  ~Student();
 private:
  long facnom;
  double usp;
};
Student::Student(char *str, char * num, long facn,
  double u) : People(str, num)
...
Student::~Student()
{cout << "~Student(): " << endl;
 delete name;
 delete egn;
}
void Student::PrintStudent() const
...
```

е неправилна заради двойното освобождаване на динамична памет. Някои дефиниции са пропуснати, тъй като са същите като в задача 160.

### 17.4.2 Конструктор за присвояване и операторна функция за присвояване

Член-данните на обект на производен клас могат да получат стойности и чрез инициализиране чрез присвояване на друг обект или направо чрез присвояване. Това се осъществява чрез конструктора за присвояване и операторната функция за присвояване на производния клас.

В общия случай, производният клас **не** наследява от основния клас конструктора за присвояване и оператора за присвояване. Има някои изключения, на които ще се спрем по-долу.

**Конструктор за присвояване**

При конструкторите за присвояване се спазва същият принцип като при обикновените конструктори на производния и основния клас. Конструкторът за присвояване на производния клас инициализира чрез присвояване собствените член-данни на класа, а конструкторът за присвояване на основния клас инициализира наследените член-данни. Конструкторите за присвояване на производни класове се дефинират по същия начин като обикновените конструктори на производни класове. Ще напомним, че ако в клас не е дефиниран конструктор за присвояване, ролята на такъв се поема от генерирания системен *конструктор за копиране* <име_на_клас>(const <име_на_клас>&).

Ще отбележим някои случаи на използване на конструкторите за присвояване на производния и основния клас.

· **В производния клас не е дефиниран конструктор за присвояване**

Възможни са:

***а) в основния клас е дефиниран конструктор за присвояване***

В този случай компилаторът генерира служебен конструктор за копиране на производния клас <име_на_производен_клас>(const <име_на_производен_клас>&), който преди да се изпълни, **активира и изпълнява** конструктора за присвояване на основния клас. Ще отбележим, че при

обикновените конструктори този случай ще предизвика грешка, ако в основния клас няма подразбиращ се конструктор. Затова в случая се казва, че конструкторът за присвояване на основния клас се наследява от производния клас.

Ще илюстрираме казаното чрез следващата задача. Тя е обобщение на задача 160.

**Задача 161.** Да се допълни класът People от предната задача чрез конструктор за присвояване.

```
// Program Zad161.cpp
#include <iostream.h>
#include <string.h>
class People
{public:
  People(char *, char *);
  People(const People&);  // конструктор за присвояване
  void PrintPeople() const;
  ~People();   // деструктор
 private:
  char * name;
  char * egn;
};
// дефиниция на двуаргументния конструктор на класа People
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
// дефиниция на конструктора за присвояване на People
People::People(const People& p)
{name = new char[strlen(p.name)+1];
 strcpy(name, p.name);
 egn = new char[11];
```

```
 strcpy(egn, p.egn);
}
// дефиниция на метода PrintPeople
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
// дефиниция на деструктора
People::~People()
{cout << "~People()\n";
 delete name;
 delete egn;
}
// декларация на класа Student, в който не е дефиниран
// конструктор за присвояване
class Student : People
{public:
  Student(char *, char *, long, double); // обикновен конструктор
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
// дефиниция на конструктора на класа Student
Student::Student(char *str, char * num,
                 long facn, double u) : People(str, num)
{facnom = facn;
 usp = u;
}
// дефиниция на метода PrintStudent
void Student::PrintStudent() const
{PrintPeople();
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
}
// декларация на класа PStudent, в който също не е дефиниран
```

```
// конструктор за присвояване
class PStudent : public Student
{public:
  PStudent(char * = “”, char * = “”, long = 0,
           double = 0, double = 0);
  void PrintPStudent() const;
 protected:
  double tax;
};
// дефиниция на конструктора на класа PStudent
PStudent::PStudent(char *str, char *num, long facn,
          double u, double t) : Student(str, num, facn, u)
{tax = t;
}
// дефиниция на метода PrintPStudent
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
{Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
 s1.PrintStudent();
 Student s2 = s1;
 s2.PrintStudent();
}
```

В резултат от изпълнението й се получава:

```
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
~People()
~People()
```

Инициализацията на обекта s1 се осъществява чрез конструктора Student(char*, char*, long, double), а инициализацията на обекта s2 – чрез генерирания от компилатора конструктор за *копиране* на класа Student. Конструкторът за копиране на класа Student се обръща към конструктора за присвояване на класа People (с аргумент s1). В резултат данните name и egn на s1 се копират в съответните полета на обекта s2. След това се изпълнява тялото на “служебния” конструктор за копиране на класа Student, при което числовите полета facnom и usp на s2 се инициализират със стойностите 42444 и 6 на съответните полета на s1.

*б) в основния клас не е дефиниран конструктор за присвояване*

В този случай се генерират “служебни” конструктори за копиране за двата класа. Конструкторът за копиране на производния клас активира конструктора за копиране на основния клас.

Нека се върнем към решението на предходната задача. Класът PStudent е производен на класа Student и в двата класа не са дефинирани конструктори за присвояване. В резултат от изпълнението на функцията:

```
void main()
{PStudent s1("Ivan Ivanov", "8206123422", 42444, 6.0, 350);
 s1.PrintPStudent();
 PStudent s2 = s1;
 s2.PrintPStudent();
}
```

се получава:

```
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
Tax: 350
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
Tax: 350
```

~People()
~People()

Инициализацията на обекта s1 се осъществява чрез конструктора PStudent(char*, char*, long, double, double), а инициализацията на обекта s2 – чрез генерирания от компилатора конструктор за копиране на класа PStudent. Конструкторът за копиране на класа PStudent извиква конструктора за копиране на класа Student, който пък извиква конструктора за присвояване на класа People (с аргумент s1). В резултат данните name и egn на s1 се присвояват на съответните полета на обекта s2, съгласно указания, в дефиницията на конструктора за присвояване на класа People, начин. След това се изпълнява тялото на конструктора за копиране на класа Student, при което полетата facnom и usp на s2 се инициализират с 42444 и 6 съответно и накрая се изпълнява тялото на конструктора за копиране на класа PStudent, при което полето tax се инициализира с 350.

· **В производния клас е дефиниран конструктор за присвояване**

В този случай отначало се активира конструкторът за присвояване на производния клас. В неговия инициализиращ списък може да има или да няма обръщение към конструктор (за присвояване или обикновен) на основния клас. Препоръчва се в инициализиращия списък на производния клас да има обръщение към конструктора за присвояване на основния клас, ако такъв е дефиниран. Ако не е указано обръщение към конструктор на основния клас, инициализирането на наследените членове става чрез подразбиращия се конструктор на основния клас. Ако основният клас няма такъв, ще се съобщи за отсъствието на подходящ конструктор.

Така конструкторът за присвояване на производния клас чрез дефиницията си определя как точно ще се инициализира наследената част.

**Задача 162.** Да се допълни и класът Student от предната задача с конструктор за присвояване.

В случая това не е необходимо, защото генерирания от компилатора служебен конструктор за копиране напълно ни устройва. Предложеното

решение е заради технически съображения. Някои дефиниции на методи са пропуснати, тъй като са аналогични на тези от задача 161.

```
// Program Zad162.cpp
#include <iostream.h>
#include <string.h>
class People
{public:
  People(char *, char *);
  People(const People&);
  void PrintPeople() const;
  ~People()
 private:
  char * name;
  char * egn;
};
People::People(char *str, char *num)
{...
}
People::People(const People& p)
{...
}
void People::PrintPeople() const
{...
}
People::~People()
{...
}
class Student : People
{public:
  Student(char *, char *, long, double);
  Student(const Student& st);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
```

```
Student::Student(char *str, char * num,
                 long facn, double u) : People(str, num)
{...
}
// дефиниция на конструктора за присвояване на Student
Student::Student(const Student& st) : People(st)
{facnom = st.facnom;
 usp = st.usp;
}
void Student::PrintStudent() const
{...
}
void main()
{Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
 s1.PrintStudent();
 Student s2 = s1;
 s2.PrintStudent();
}
```

Забележете, аргументът на обръщението People(st), в инициализиращия списък на конструктора за присвояване на класа Student, е от тип const Student&, а не const People& (В т.17.5 ще разгледаме преобразуването на типовете).

Инициализацията на обекта s1 се осъществява чрез конструктора Student(char*, char*, long, double), а инициализацията на обекта s2 – чрез конструктора за присвояване Student(const Student&). При изпълнението на инициализацията на s2 отначало се изпълнява конструкторът за присвояване на класа People (с аргумент s1). В резултат данните name и egn на s1 се инициализират със съответните от обекта s2. След това се изпълнява самият конструктор за присвояване на класа Student, при което полетата facnom и usp на s2 се инициализират с 42444 и 6 съответно.

Друга реализация на конструктора за присвояване на класа Student е:

```
Student::Student(const Student& st) : People(st.name, st.egn)
{facnom = st.facnom;
 usp = st.usp;
```

```
}
```

*ако член-данните name и egn на класа People са обявени в секция protected.* В този случай за инициализиране на наследените член-данни е използван двуаргументният конструктор на People.

Ако добавим подразбиращ се конструктор на класа People, т.е.

```
People::People()
 {name = new char[1];
  strcpy(name, "");
  egn = new char[1];
  strcpy(egn, "");
}
```

и конструкторът за присвояване на класа Student има вида:

```
Student::Student(const Student& st)
{facnom = st.facnom;
 usp = st.usp;
}
```

след изпълнението на фрагмента

```
Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
Student s2 = s1;
s2.PrintStudent();
```

полетата name и egn на s2 се инициализират с празния низ, а facnom и успех с 42444 и 6 съответно.

Причината е, че в инициализиращия списък на конструктора за присвояване на класа Student не е указан начинът за инициализиране на член-данните на основния клас. В този случай инициализацията се осъществява чрез конструктора по подразбиране на основния клас People.

Ще дадем още един пример.

```
#include <iostream.h>
class base
{public:
 base(int x = 99)
 {b = x;
 }
 void f() const
```

```
 {cout << "base: b: " << b << endl;
 }
 private:
  int b;
};
class der : public base
{public:
der(int x = 11) : base(x)
{d = x;
}
der(const der& p)
{d = p.d + 5;
}
void f() const
{base::f();
 cout << "der: d: " << d << endl;
}
private:
 int d;
};
void main()
{der d1;
 d1.f();
 der d2 = d1;
 d2.f();
}
```

В резултат от изпълнението му се получава:

```
base: b: 11
der: d: 11
base: b: 99
der: d: 16
```

В него основният клас не предефинира конструктора за копиране. В производния клас има конструктор за присвояване, в който явно не е казано как точно става инициализацията на наследената компонента. Тъй като в base има конструктор по подразбиране, инициализацията се осъществява чрез него. Ако променим конструктора на класа base в:

```
base(int x)
{b = x;
}
```

и конструкторът за присвояване на der стане:

```
der(const der& p)
{d = p.d + 5;
}
```

компилаторът ще съобщи за отсъствието на подходящ конструктор на класа base, тъй като конструктор по подразбиране за класа base не съществува.

**Операторна функция за присвояване**

Операторната функция за присвояване на производен клас трябва да указва как да стане присвояването както на собствените, така и на наследените си член-данни. За разлика от конструкторите на производни класове тя прави това в тялото си (не поддържа инициализиращ списък). Използването й зависи от това дали такава е дефинирана в производния клас. Ще напомним, че ако в клас не е дефинирана операторна функция за присвояване, компилаторът създава operator=(const <име_на_клас>&).

Ще разгледаме следните два случая:

· **В производния клас не е дефинирана операторна функция за присвояване**

Компилаторът създава операторна функция за присвояване на производния клас. **Тя се обръща и изпълнява операторната функция за присвояване на основния клас**, чрез която инициализира наследената част, след това инициализира чрез присвояване и собствената част на производния клас. Затова в този случай се казва, че операторът за присвояване на основния клас се наследява, т.е. за наследените член-данни се използва подразбиращият се или предефинираният оператор за присвояване на основния клас.

· **В производния клас е дефинирана операторна функция за присвояване**

Дефинираният в производния клас оператор за присвояване **трябва да се погрижи за наследените компоненти. Налага се в тялото на неговата дефиниция да има обръщение към дефинирания оператор за присвояване на**

**основния клас**, ако има такъв. Ако това не е направено **явно**, ***стандартът на езика не уточнява как ще стане присвояването на наследените компоненти***. В случая се казва, че операторът за присвояване на основния клас не се наследява.

**Пример:** Да разгледаме следната йерархична схема от 4 класа: базов клас base и три негови наследници: der1, der2 и der3, реализирана чрез програмата:

```
#include <iostream.h>
class base
{public:
 base(int x = 0)
 {b = x;
 }
 base& operator=(const base &x)
 {if(this!=&x) b = x.b + 1;
  return *this;
 }
 protected:
  int b;
};
class der1 : public base
{public:
 der1(int x = 1)
 {d = x;
 }
 der1& operator=(const der1& x)
 {if(this!=&x)
  {d = x.d + 2;
   b = x.b + 3;
  }
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
```

```
  int d;
};
class der2 : public base
{public:
 der2(int x = 2)
 {d = x;
 }
 der2& operator=(const der2& x)
 {if(this !=&x)
  {d = x.d + 3;
  }
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
class der3 : public base
{public:
 der3(int x = 3)
 {d = x;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
void main()
{der1 d11(5), d12;
 der2 d21(5), d22;
 der3 d31(5), d32;
 d12 = d11;
 d22 = d21;
```

```
    d32 = d31;
    cout << "d11: "; d11.Print();
    cout << "d12: "; d12.Print();
    cout << "d21: "; d21.Print();
    cout << "d22: "; d22.Print();
    cout << "d31: "; d31.Print();
    cout << "d32: "; d32.Print();
   }
```

В резултат от изпълнението й се получава:

```
   d11: der: 5  base: 0
   d12: der: 7  base: 3
   d21: der: 5  base: 0
   d22: der: 8  base: 0
   d31: der: 5  base: 0
   d32: der: 5  base: 1
```

Класовете der1, der2 и der3 са дефинирани и използвани по идентичен начин с изключение на предефинирания оператор за присвояване. В класа der1 операторът за присвояване е предефиниран и се грижи за наследената част. В класа der2 операторът за присвояване е предефиниран, но не указва как става присвояването на наследената член-променлива. Тъй като операторът за присвояване на базовия клас в този случай не се наследява, стандартът на езика не уточнява стойността на наследената член-променлива на обекта d22. В този случай нейната стойност е тази от инициализацията der2 d22. В класа der3 операторът за присвояване не е предефиниран. Тогава за собствените на класа компонента се използва подразбиращият се, а за наследената – предефинираният оператор за присвояване на базовия клас се наследява и изпълнява.

**Задача 163.** Да се допълнят класовете People и Student от предишната задача с операторни функции за присвояване.

```
   // Program Zad163.cpp
   #include <iostream.h>
   #include <string.h>
   class People
```

```
{public:
  People(char * = "", char * = "");
  People(const People&);
  People& operator=(const People& p);
   void PrintPeople() const;
   ~People();
 private:
  char * name;
  char * egn;
};
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
People::People(const People& p)
{name = new char[strlen(p.name)+1];
 strcpy(name, p.name);
 egn = new char[11];
 strcpy(egn, p.egn);
}
People& People::operator=(const People& p)
{if(this!=&p)
 {delete name;
  delete egn;
  name = new char[strlen(p.name)+1];
  strcpy(name, p.name);
  egn = new char[11];
  strcpy(egn, p.egn);
}
 return *this;
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
```

```
}
People::~People()
{delete name;
 delete egn;
}
class Student : People
{public:
  Student(char * = "", char * = "", long = 0, double = 0);
  Student(const Student& st);
  Student& operator=(const Student& st);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
Student::Student(char *str, char * num,
                 long facn, double u) : People(str, num)
{facnom = facn;
 usp = u;
}
Student::Student(const Student& st) : People(st)
{facnom = st.facnom;
 usp = st.usp;
}
Student& Student::operator=(const Student& st)
{if(this!=&st)
{People::operator=(st);
 facnom = st.facnom;
 usp = st.usp;
 }
 return *this;
}
void Student::PrintStudent() const
{PrintPeople();
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
```

```
}
// дефиниране на клас PStudent
class PStudent : public Student
{public:
  PStudent(char * = "", char * = "", long = 0,
  double = 0, double = 0);
  PStudent& operator=(const PStudent& st);
  void PrintPStudent() const;
 protected:
  double tax;
};
PStudent::PStudent(char *str, char *num, long facn,
          double u, double t) : Student(str, num, facn, u)
{tax = t;
}
PStudent& PStudent::operator=(const PStudent& st)
{if(this!=&st)
{Student::operator=(Student(st));
  tax = st.tax;
 }
 return *this;
}
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
{PStudent s1("Ivan Ivanov", "8206123422", 42444, 6.0, 4444);
 s1.PrintPStudent();
 PStudent s2("Jonko Dimov", "9012074442", 43344, 5, 3434);
 s2.PrintPStudent();
 s2 = s1;
 s2.PrintPStudent();
}
```

Ще дадем още едно решение на задачата, което смятаме за полезно.

```
#include <iostream.h>
```

```
#include <string.h>
class People
{public:
  People(char * = "", char * = "");
  People(const People&);
  People& operator=(const People& p);
  ~People();
  void PrintPeople() const;
  void del();
  void CopyPeople(const People& p);
 private:
  char * name;
  char * egn;
};
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
People::People(const People& p)
{CopyPeople(p);
}
People& People::operator=(const People& p)
{if(this!=&p)
{del();
 CopyPeople(p);
}
 return *this;
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
People::~People()
{del();
```

```
}
void People::del()
{delete name;
 delete egn;
}
void People::CopyPeople(const People& p)
{name = new char[strlen(p.name)+1];
 strcpy(name, p.name);
 egn = new char[11];
 strcpy(egn, p.egn);
}
class Student : People
{public:
  Student(char * = "", char * = "", long = 0, double = 0);
  Student(const Student& st);
  Student& operator=(const Student& st);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
Student::Student(char *str, char * num,
             long facn, double u) : People(str, num)
{facnom = facn;
 usp = u;
}
Student::Student(const Student& st) : People(st)
{facnom = st.facnom;
 usp = st.usp;
}
Student& Student::operator=(const Student& st)
{if(this != &st)
{del();
 CopyPeople(st);
 facnom = st.facnom;
 usp = st.usp;
```

```
}
 return *this;
}
void Student::PrintStudent() const
{PrintPeople();
 cout << "Fac. nomer: " << facnom << endl;
 cout << "Uspeh: " << usp << endl;
}
class PStudent : public Student
{public:
  PStudent(char * = "", char * = "", long = 0,
           double = 0, double = 0);
  PStudent(const PStudent& st):Student(st),tax(st.tax)
  {}
  PStudent& operator=(const PStudent& st);
  void PrintPStudent() const;
 protected:
  double tax;
};
PStudent::PStudent(char *str, char *num, long facn,
              double u, double t) : Student(str, num, facn, u)
{tax = t;
}
PStudent& PStudent::operator=(const PStudent& st)
{if (this != &st)
 {Student p = st;
  Student::operator=(Student(st));
  tax = st.tax;
 }
 return *this;
}
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
```

```
{PStudent s1("Ivan Ivanov", "8206123422", 42444, 6.0, 500);
 s1.PrintPStudent();
 PStudent s2("Jonko Dimov", "9012074442", 43333, 4.0, 700);
 s2.PrintPStudent();
 s2 = s1;
 s2.PrintPStudent();
}
```

## 17.5 Преобразуване на типовете

Ако основният клас, който се наследява от производния клас е с атрибут **public**, възможно е взаимно заменяне на обекти от двата класа. Заменянето може да се извършва при инициализиране, при присвояване и при предаване на параметри на функции. Могат да се заменят обекти, псевдоними на обекти, указатели към обекти и указатели към методи. Замяната в посока “производен с основен” се счита за безопасна, докато замяната в обратната посока “основен с производен” може да предизвика проблеми.

Процесът на замяна е свързан с преобразувания, които за различните случаи са явни или неявни.

За да покажем тези преобразувания ще използваме класовете base и der, дефинирани по следния начин:

```
class base
{public:
 base(int x = 0)
 {b = x;
 }
 int get_b() const
 {return b;
 }
 void f()
 {cout << "b: " << b << endl;
 }
 private:
  int b;
 };
```

```
class der : public base
{public:
 der(int x = 0) : base(x)
 {d = 5;
 }
 int get_d() const
 {return d;
 }
 void f_der()
 {cout << "class der: d: " << d
       << " b: " << get_b() << endl;
 }
 private:
  int d;
 };
```

### 17.5.1 Преобразуване в посока “производен с основен”

Обект, псевдоним на обект или указател към обект на производен клас се преобразуват съответно в обект, псевдоним на обект или указател към обект на основен клас чрез ***неявни стандартни преобразувания***. На практика тези преобразувания се свеждат до използване само на наследените компоненти на класа. Последното се основава на факта, че производният клас наследява всички свойства на базовия клас и може да бъде използван вместо него.

**Пример:** В резултат от изпълнението на фрагмента:

```
der d; d.f_der();
 base x = d; x.f();
 der &d1 = d; d1.f_der();
 base &y = d1; y.f();
 der *d2 = &d; (*d2).f_der();
 base *z = d2; (*z).f();
```

се получава

```
class der: d: 5 b: 0
b: 0
class der: d: 5 b: 0
```

```
b: 0
class der: d: 5 b: 0
b: 0
```

### 17.5.2 Преобразуване в посока “основен производен”

Тъй като основният клас не съдържа собствените компоненти на производния клас, това преобразуване се осъществява само чрез явно указване.

**Най-често се срещат следните случаи:**

*а) Присвояване и инициализиране на обект от производен клас с обект на основен клас*

Нека x е обект на класа base, а y е обект на производния му клас der. Искаме на y да присвоим x. Стандартно се реализира чрез явно преобразуване на x в обект на клас der, т.е.

```
base x;
der y = (der)x;
```

Операцията е опасна, тъй като собствените компоненти на обекта y ще останат неинициализирани и *опитът за промяната им може да доведе до сериозни последици*. Затова някои реализации на езика, в това число и Visual C++ 6.0, **не реализират това преобразувание**.

Подобна е ситуацията при използване на указатели към обекти:

```
base *pb = new base;
der* pd = (der*) pb;
```

Извършва се явно преобразуване на pb в указател към обект на клас der. Указателят pd към обект на der не сочи към *истински обект* от клас der. Областта в паметта, свързана с указателя pd, няма собствени компоненти на класа der. Опитът за използването им **може** да предизвика сериозни проблеми, тъй като ще се използва памет, която е определена за други цели. Някои реализации на езика не реализират това преобразуване. Visual C++ 6.0 го извършва. От примера по-долу се вижда, че се извежда случайна стойност, но опитът за промяна на тази памет, **може** да е с непредвидими за програмата последици. Дефиниция на

указателя pd към der може да се използва за извикване на собствена член-функция на der.

**Пример:** Изпълнението на фрагмента:

```
base *pb = new base;
(*pb).f();     // или pb->f();
der *pd = (der*) pb;
(*pd).f_der();    // или pd->f_der();
cout << pd->get_b() << endl;
```

води до следния резултат:

```
b: 0
class der: d: -33686019 b: 0
0
```

*б) Присвояване на указател към метод на основен клас на указател към метод на производен клас*

Чрез дефиницията

```
void (base::*pb)() = base::f;
```

pb се обявява за указател към метода f на класа base, който няма параметри и е от тип void. За да се използва този указател е необходимо той да се свърже с конкретен обект, т.е.

```
base x;
(x.*pb)();
```

В резултат се осъществява обръщение към метода f чрез указателя pb към него и се получава:

```
b: 0
```

Чрез дефиницията

```
void (der::*pd)()
```

pd се определя като указател към метод на производния клас der като методът е без параметри и е от тип void.

*Възниква въпросът:* Може ли указателят pb към метод на основния клас да се присвои на указателя pd, т.е. може ли да запишем:

```
void (der::*pd)() = pb;
der y(20);
(y.*pd)();
```

Отговорът е положителен. При присвояването ще се извърши неявно присвояване на типовете. Обектът y на производния клас der съдържа

наследената част от основния клас base и метода f() в частност. Затова указателят pd ще указва правилно. В резултат се получава:

```
b: 0
b: 20
```

Обратното присвояване – указател към член-функция на производния клас да се присвои на указател към член-функция на основен клас изисква явно преобразуване и дали ще се използва правилно зависи единствено от програмиста.

**Пример:** Фрагментът:

```
void (der::*pd)() = der::f_der;
void (base::*pb)() = pd;
```

е недопустим. Проблемът идва от това, че pb е указател към метод на base от тип void и без параметри. В същото време pd е указател към член-функцията f_der на der също от тип void и без параметри, но f_der има достъп до собствените членове на der, които не са членове на base.

Допустимо е присвояването:

```
void (base::*pb)();
pb = (void (base::*)()) der::f_der; //явно преобразуване
```

но използването му може да доведе до грешка или до нееднозначност, както е показано в примера по-долу.

**Пример:**

```
base x(5);
der y(10);
base *pf = &y; // pf сочи към обект на класа der
(pf->*pb)();   // извикване на метода der::f_der
pf = &x;       // pf сочи към обекта x на класа base
(pf->*pb)();   // грешка, зависи от реализацията
```

*в) Достъп до собствени членове на производния клас чрез обект на основния клас*

Такъв достъп директно не е възможен. Непряк достъп е възможен и се осъществява чрез указатели и преобразувания.

**Пример:**

```
der y;
der *pd = &y;  // инициализация на указателя pd към обект на der
```

```
base *pb = pd; // неявно преобразуване
```

Указателите pb и pd сочат към обекта y на класа der, но са различни. Компилаторът проверява допустимостта на използването им в зависимост **за кой клас се отнася.**

Обръщението

```
pd->f_der();
```

е допустимо и активира, чрез указателя pd към обекта y, метода f_der(), но обръщението

```
pb->f_der();
```

е недопустимо, тъй като pb е указател към base, който няма f_der() за свой метод. За да стане възможно последното трябва да се използва явно преобразуване

```
((der *) pb)->f_der();
```

Ще напомним, че казаното се отнася само за производни класове с атрибут за област public на основния клас. За производни класове с атрибути за област private и protected не са дефинирани подобни преобразувания. Това е естествено, тъй като в тези случаи производният клас не притежава всички свойства на базовия и не може да го замести.

## 17.6 Шаблони на класове и наследяване

Използването на шаблони на класове и наследяването може да се разгледа в следните случаи:

· **шаблон на клас – наследник на обикновен клас**

В този случай, ако base е обикновен клас, т.е.

```
class base
{<тяло>
};
```

декларацията на производния на base шаблон на клас der ще има вида:

```
template <class T>
class der:<атрибут_за_област> base
{<тяло>
};
```

· **обикновен клас – наследник на шаблон на клас**

Ако tempbase е шаблон на основен клас

```
template <class T>
class tempbase
{<тяло>
};
```

декларацията на обикновен производен клас на шаблона tempbase трябва да задава стойност на параметъра за тип на tempbase, например int в случая и има вида

```
class der:<атрибут_за_област> tempbase<int>
{<тяло>
};
```

· **шаблон на клас – наследник на шаблон на клас**

Нека е декларирана шаблон на клас tempbase с параметър T:

```
template <class T>
class tempbase
{<тяло>
};
```

Декларацията на производен шаблон на клас на шаблона tempbase може да стане по два начина. При първия, производният шаблон на клас запазва същия параметър за тип като базовия, т.е.

```
template <class T>
class tempder1: атрибут_за_област tempbase<T>
{<тяло>
};
```

В този случай на всеки възможен базов клас съответства точно един производен. При втория начин, производният шаблон на клас въвежда и други параметри за тип:

```
template <class T, class U, ...>
class tempder2 : атрибут_за_област tempbase<T>
{<тяло>
};
```

При тази декларация на всеки възможен базов клас съответства фамилия от производни класове.

**Задача 164.** Да се дефинира шаблон на клас “точка в равнината” с параметър T за тип (тип на координатите) и клас, определящ отсечка в равнината с краища – точки с цели координати. На тези класове да се дефинират шаблони на производни класове, определящи “точка в равнината с цвят” и “отсечка в равнината с цвят”.

Програма Zad164.cpp рашава задачата. Цветът на точка (отсечка) се задава с числов параметър.

```
// Program Zad164.cpp
#include <iostream.h>
// дефиниция на шаблон на клас Point, определящ точка
// в равнината с координати от тип T
template <class T>
class Point
{public:
 Point(T = 0, T = 0);
 void print() const;
 private:
  T x, y;
 };
template <class T>
Point<T>::Point<T>(T a, T o)
{x = a;
 y = o;
}
template <class T>
void Point<T>::print() const
{cout << "Point(" << x << ", " << y << ")\n";
}
// дефиниция на клас Segm, опреелящ отсечка в равнината
// с краища – точки с цели координати
class Segm
{public:
 Segm(int a1=0, int o1=0,int a2=0, int o2=0);
 void print() const;
 private:
```

```
  int x1, y1, x2, y2;
 };
Segm::Segm(int a1, int o1,int a2, int o2)
{x1 = a1;
 y1 = o1;
 x2 = a2;
 y2 = o2;
}
void Segm::print() const
{cout << "Segm(" << x1 << ", " << y1 << ")"
      <<"(" << x2 << ", " << y2 << ")\n";
}
// дефиниция на клас ColPoint, определящ точка
// с реални координати и цвят
class ColPoint : public Point<double>
{public:
 ColPoint(double = 0, double = 0, int = 0);
 void print() const;
 private:
  int col;
};
ColPoint::ColPoint(double a, double o, int c) :
                                                Point<double>(a, o)
{col = c;
}
void ColPoint::print() const
{Point<double>::print();
 cout << "Color: " << col << endl;
}
// дефиниция на шаблон на клас ColPoint1, наследник на шаблона
// на класа Point и запазващ параметъра на шаблона на Point
template <class T>
class ColPoint1 : public Point<T>
{public:
 ColPoint1(T = 0, T = 0, T = 0);
 void print() const;
```

```
 private:
  T col;
 };
template <class T>
ColPoint1<T>::ColPoint1<T>(T a, T o , T c) : Point<T>(a, o)
{col = c;
}
template <class T>
void ColPoint1<T>::print() const
{Point<T>::print();
 cout << "color(col1): " << col << endl;
}
// дефиниция на шаблон на клас ColPoint2 с два параметъра,
// наследник на шаблона на класа Point. Параметри:
// T – тип на координатите на точката
// U – тип на цвета
// на класа Point и запазващ параметъра на шаблона на Point
template <class T, class U>
class ColPoint2: public Point<T>
{public:
 ColPoint2(T a = 0, T o = 0, U c = 0);
 void print() const;
 private:
 U col;
};
template <class T, class U>
ColPoint2<T, U>::ColPoint2<T, U>(T a, T o, U c) : Point<T>(a, o)
{col = c;
}
template <class T, class U>
void ColPoint2<T, U>::print() const
{Point<T>::print();
 cout << "color(col2): " << col << endl;
}
template <class T>
class ColSegm : public Segm
```

```
{public:
 ColSegm(int a1=0, int o1=0, int a2=0, int o2=0,
   T c=0) : Segm(a1,o1,a2,o2)
 {col = c;
 }
 void print()
 {Segm::print();
  cout << "Color: " << col << endl;
 }
 private:
 T col;
};
```

В резултат от изпълнението й се получава:

```
Point(1.5, 3,5)
Segm(1,1) (5.5)
Point(3.8, -4.5)
Color: 8
Point(5, 10)
Color(col1): 15
Point(1.4, 3.5)
Color(col2): 7
Segm(0, 0) (1,1)
Color: 3.4
```

## 17.7 Компилация и свързване

Случаят, когато дефинициите на основния, на производния клас и програмата се намират в един файл, е тривиален. Файлът се компилира и свързва самостоятелно в готов за изпълнение програмен модул.

Когато декларациите на основния и производния клас, дефинициите на функциите им и текста на програмата, ползваща производния клас са разположени в различни файлове, компилацията и свързването се осъществяват на следните стъпки:

- Компилиране на декларацията на основния клас и дефинициите на функциите му;

- Компилиране на декларацията на производния клас и дефинициите на член-функциите му, заедно с декларацията на основния клас;
- Компилиране на програмата заедно с декларациите на основния и производния клас;
- Свързване на получените обектни модули.

## Допълнителна литература


1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.

# 18

# Множествено наследяване. Виртуални класове и функции. Полиморфизъм

## 18.1 Множествено наследяване

В случаите, когато производният клас наследява няколко основни класа, се казва, че класът е с множествено наследяване. Този вид наследяване е мощен инструмент на ООП, тъй като чрез него се изграждат графовидни йерархични структури.

В параграф 17 дефинирахме производен клас. Сега ще конкретизираме дефиницията за случая на множественото наследяване.

Дефиницията на производен клас се състои от декларация на класа и дефиниции на методите му.

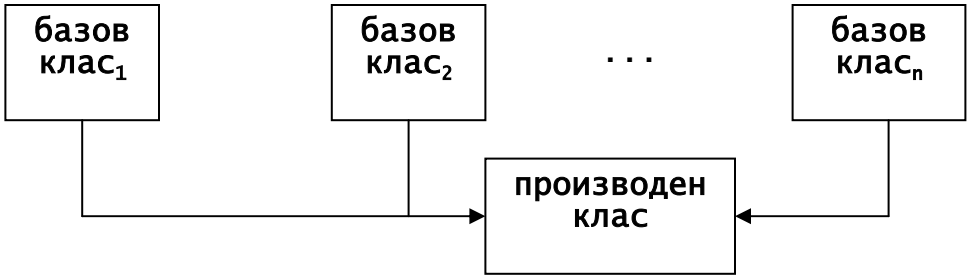
**Декларираме на производен клас с множествено наследяване**

<декларация_на_производен_клас> ::=
class <име_на-производен_клас> :
[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас$_1$>,
[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас$_2$>{,
[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас$_n$>}$_{опц}$
{<декларация_на_компоненти>
};
<име_на-производен_клас> ::= <идентификатор>
<атрибут_за_област> ::= public | private | protected
<име_на_базов_клас$_i$> ::= <идентификатор>

18.1 Декларираме на производен клас с множествено наследяване

Атрибутите за област се задават за всеки базов клас. Семантиката им беше вече пояснена при разглеждане на производни класове с единично наследяване. Ако за някои клас атрибутът за област е пропуснат, подразбира се private.

Определената на фиг. 18.1 декларация задава следната йерархия:




Производният клас наследява компонентите на всички базови класове като видът на наследяване private, public или protected се определя от атрибута за област на базовия клас. Правилата са същите като при единичното наследяване.

За член-функциите на голямата четворка на производен клас с множествено наследяване са в сила същите правила, като при производен клас с единично наследяване. В общия случай тези член-функции за основните класове не се наследяват от производния им клас. Изключенията отново са при конструкторите за присвояване и операторните функции за присвояване.

Ще напомним дефиницията на конструктора на производен клас с множествено наследяване (Фиг. 18.2).

**Дефиниция на конструктор на производен клас**

```
<име_на_производен_клас>::<име_на_производен_клас>>(<параметри>)
         <инициализиращ_списък>
{<тяло>
}
<инициализиращ_списък> ::= <празно>|
                  : <име_на_основен_клас>(<параметри_i>)
                  {,<име_на_основен_клас>(<параметри_i>)}_опц
<параметри> ::= <празно>|
                <параметър>|
```

<параметри>, <параметър>

<параметър> ::= <тип> <име_на_параметър>

<име_на_параметър> ::= <идентификатор>

<параметри$_i$> е конструкция, която има синтаксиса на <фактически_параметри> от дефиницията на обръщение към функция (глава 8), а <тяло> се определя като тялото на който и да е конструктор.

В <инициализиращ_списък> може да има повече от едно обръщение към конструктор на основен клас.

Фиг. 18.2 Конструктор на производен клас

При извикването на този конструктор последователно се извикват:

а) Конструкторите на базовите класове по реда на тяхното задаване не в инициализиращия списък на конструктора, а в декларацията на производния клас.

б) Конструкторите на класовете, чиито обекти са членове на производния клас. Редът на извикване съответства на реда на декларираме на тези членове в тялото на производния клас.

в) Конструкторът на производния клас.

**Отново са възможни:**

***1) В основен клас не е дефиниран конструктор***

В този случай в инициализиращия списък на конструктора на производния клас не се прави обръщение към конструктора на този клас и наследената му част остава неинициализирана.

***2) В основен клас има един конструктор с параметър, от който не следва подразбиращия се***

Тогава ако в производния клас е дефиниран конструктор, в инициализиращия му списък задължително трябва да има обръщение към конструктора с параметър на този основен клас. Ако в производния клас не е дефиниран конструктор за присвояване, компилаторът ще сигнализира грешка.

***3) Основен клас има няколко конструктора в т. число подразбиращ се***

Ако в производния клас е дефиниран конструктор, в инициализиращия му списък може да не се посочва конструктор за този основен клас. Ще се използва подразбиращия се. Ако в производния клас не е дефиниран конструктор, компилаторът автоматично създава за него подразбиращ се

конструктор. В този случай обаче всички основни класове на производния клас трябва да имат подразбиращ се конструктор.

Дефинирането на деструкторите става по описания вече начин. Всеки деструктор се грижи само за собствените си компоненти. Извикването им става в обратен ред – първо се извиква деструкторът на производния клас, след това, в обратен ред, се извикват деструкторите на класовете на обектите – членове на производния клас и най-накрая на основните му класове, отново в обратен ред на реда на извикване на техните конструктори.

**Пример:** Резултатът от изпълнението на програмата:

```
#include <iostream.h>
class base1
{public:
 base1(int x = 0)
 {cout << "base1:\n";
  b1 = x;
 }
 ~base1()
 {cout << "~base1()\n";
 }
 private:
  int b1;
 };
class base2
{public:
 base2(int x = 0)
 {cout << "base2:\n";
  b2 = x;
 }
 ~base2()
 {cout << "~base2()\n";
 }
 private:
  int b2;
 };
```

```
class base3
{public:
 base3(int x = 0)
 {cout << "base3:\n";
  b3 = x;
 }
 ~base3()
 {cout << "~base3()\n";
 }
 private:
  int b3;
 };
class der : public base2, base1, base3
{public:
 der(int x = 0) : base3(x), base1(x), base2(x)
 {d = 5;
 }
 private:
  int d;
 };
void main()
{der d1(5);
}
```

е

```
base2
base1
base3
~base3()
~base1()
~base2()
```

Същият е резултатът от изпълнението на програмата ако от инициализиращия списък на класа der пропуснем някое от обръщенията към основните класове base1, base2 или base3, или даже всичките. В тези случаи се използват конструкторите по подразбиране на основните класове. Резултатът от изпълнението не се променя ако в производния клас der не е дефиниран конструктор. В този случай компилаторът

създава за der конструктор по подразбиране, който се обръща към конструкторите по подразбиране на основните класове в реда, указан в декларацията на производния клас.

В тази програма класът der има две собствени и 3 наследени компоненти (конструкторите и деструкторите не се наследяват). Може да си мислим за него като клас от вида:

```
class der : public base2, base1, base3
{public:
 der(int x = 0) : base3(x), base1(x), base2(x)
 {d = 5;
 }
private:
  int d;
  int b1, b2, b3;
};
```

Дефиницията der d1(5); предизвиква създаване на обект d1 с компоненти от вида:

| | | |
|---|---|---|
| d | - | 0x0065FDE8 |
| b3 | - | 0x0065FDE4 |
| b1 | - | 0x0065FDE0 |
| b2 | - | 0x0065FDDC |

и извикване на конструктора на класа der с параметър 5. Преди да се изпълни неговото тяло се изпълняват конструкторите на базовете класове base2, base1 и base3 с параметър 5. Това инициализира отделената памет за член-данните на d1 с 5.

Дефинирането на конструктора за присвояване и операторната функция за присвояване на производен клас с множествено наследяване се извършва по същия начин както при единичното наследяване.

Ще напомним, че ако в производния клас не е дефиниран конструктор за присвояване, компилаторът генерира за него “служебен” конструктор за копиране, който преди да се изпълни активира и изпълнява конструкторите за присвояване (копиране) на всички основни класове в

реда, указан в декларацията на производния клас. Ако в производния клас е дефиниран конструктор за присвояване, препоръчва се в инициализиращия му списък да има обръщения към конструкторите за присвояване на основните класове (ако такива са дефинирани). Ако за някои основен клас не е указано такова обръщение, а е указан обикновен негов конструктор, инициализирането на наследените компоненти на този клас става чрез указания конструктор. Ако не е указано обръщение към конструктор, използва се конструкторът по подразбиране на основния клас, ако такъв съществува или се съобщава за отсъствието на подходящ конструктор за този основен клас, ако не съществува конструктор по подразбиране.

Операторната функция за присвояване на производен клас с множествено наследяване има вида:

```
<производен_клас>&
      <производен_клас>::operator=(const<производен_клас>& p)
{if (this != &p)
 {<основен_клас1>::operator=(p);
  <основен_клас2>::operator=(p);
    ...
  <основен_класN>::operator=(p);
    // дефиниране на присвояването
    // за собствените за класа компоненти
    Del();  // изтриване от ДП на собствените компоненти на
            // подразбиращия се обект
   Copy(p); // копиране на собствените компоненти на p в
            // подразбиращия се обект
 }
 return *this;
}
```

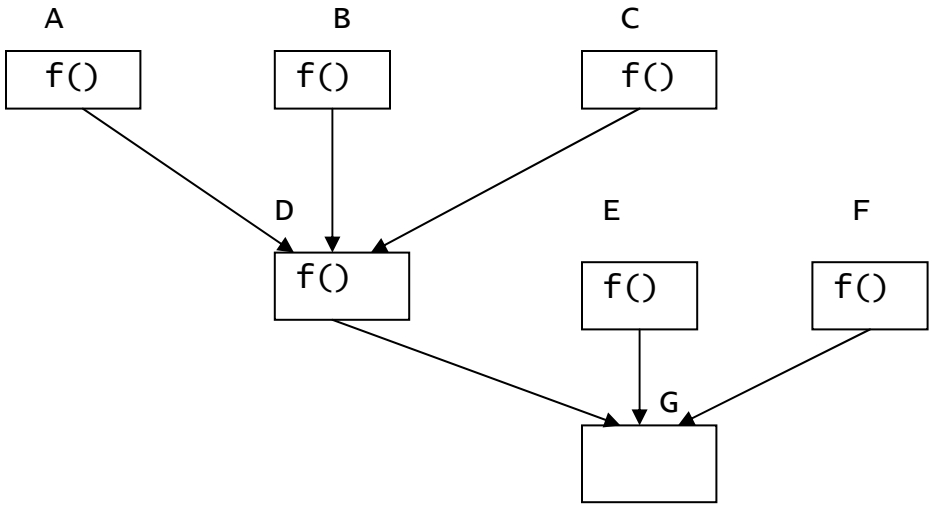
Ако в производния клас не е дефинирана операторна функция за присвояване, компилаторът създава такава. Тя изпълнява операторните функции за присвояване на всички основни класове на производния клас. Ако в производния клас е дефинирана операторна функция за присвояване, тя трябва да се погрижи за присвояването на наследените компоненти. Ако това не е направено явно, стандартът на езика не уточнява как ще стане присвояването на наследените компоненти.

Чрез обект на производния клас могат да се викат всички негови public компоненти – собствени и наследени. Ако в базовите класове са дефинирани компоненти с еднакви имена, необходимо е да се използва пълното име на компонентата, т.е. <име_на_клас>::<компонента>. Ако всички основни класове $base_i$ на производния клас der имат член-функция f() и член-данна x с еднакви имена, различаването им в производния клас става чрез пълните им имена:

base1::f(), base2::f(), ...

base1::x, base2::x, ...

**Пример:** Нека имаме йерархията:




Класът G наследява метода f() на класовете A, B, C, D, E и F. В резултат от изпълнението на програмата:

```
#include <iostream.h>
class A
{public:
 A(int x = 1)
 {a = x;
 }
 void f() const
 {cout << "A: " << a << endl;
 }
 private:
  int a;
};
class B
```

```
{public:
 B(int x = 2)
 {b = x;
 }
 void f() const
 {cout << "B: " << b << endl;
 }
 private:
  int b;
};
class C
{public:
 C(int x = 3)
 {c = x;
 }
 void f() const
 {cout << "C: " << c << endl;
 }
 private:
  int c;
};
class D : public A, public B, public C
{public:
 D(int x = 4)
 {d = x;
 }
 void f() const
 {cout << "class D: \n" ;
  A::f(); B::f(); C::f();
  cout << "D: " << d << endl;
 }
 private:
  int d;
};
class E
{public:
```

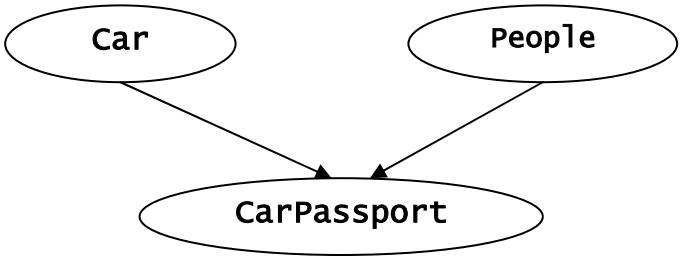
```
 E(int x = 5)
 {e = x;
 }
 void f() const
 {cout << "E: " << e << endl;
 }
 private:
  int e;
};
class F
{public:
 F(int x = 6)
 {fi = x;
 }
 void f() const
 {cout << "F: " << fi << endl;
 }
 private:
  int fi;
};
class G : public D, public E, public F
{public:
 G(int x = 7)
 {g = x;
 }
 void f() const
 {cout << "G: " << endl;
  D::f(); E::f(); F::f();
  cout << "G: " << g << endl;
 }
 private:
  int g;
};
void main()
{G ge;
 ge.A::f();
```

```
    ge.B::f();
    ge.C::f();
    ge.f();
   }
```

се получава:

```
   A: 1
   B: 2
   C: 3
   G:
   class D:
   A: 1
   B: 2
   C: 3
   D: 4
   E: 5
   F: 6
   G: 7
```

В следващата задача е реализирана следната йерархия:




**Задача 166.** Да се дефинират класове Car и People, определящи понятията “автомобил” и “човек”. Да се дефинира клас CarPassport, производен на класовете Car и People и определящ понятието “паспорт на автомобил”. За всеки от класовете да се определи голямата четворка.

Програма Zad166.cpp решава задачата.

```
// Program Zad166.cpp
#include <iostream.h>
#include <string.h>
```

```
class Car
{public:
  Car(char * = "", unsigned = 0, unsigned = 0);
  ~Car();
  Car(const Car&);
  Car& operator=(const Car&);
  void display() const;
 private:
  char *mark;
  unsigned year;
  unsigned reg_numb;
};
Car::Car(char *m, unsigned y, unsigned r_n)
{mark = new char[strlen(m)+1];
 strcpy(mark, m);
 year = y;
 reg_numb = r_n;
}
Car::~Car()
{cout << "~Car()\n";
 delete mark;
}
Car::Car(const Car& c)
{mark = new char[strlen(c.mark)+1];
 strcpy(mark, c.mark);
 year = c.year;
 reg_numb = c.reg_numb;
}
Car& Car::operator=(const Car& c)
{if (this != &c)
 {delete mark;
  mark = new char[strlen(c.mark)+1];
  strcpy(mark, c.mark);
  year = c.year;
  reg_numb = c.reg_numb;
 }
```

```
  return *this;
}
void Car::display() const
{cout << "Mark: " << mark << endl;
 cout << "Year: " << year << endl;
 cout << "Reg. Number: " << reg_numb << endl;
}
class People
{public:
  People(char * = "", char * = "");
  People(const People&);
  People& operator=(const People& p);
  void display() const;
  ~People();
 private:
  char * name;
  char * egn;
};
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
People::People(const People& p)
{name = new char[strlen(p.name)+1];
 strcpy(name, p.name);
 egn = new char[11];
 strcpy(egn, p.egn);
}
People& People::operator=(const People& p)
{if (this != &p)
 {delete name;
  delete egn;
  name = new char[strlen(p.name)+1];
  strcpy(name, p.name);
```

```
  egn = new char[11];
  strcpy(egn, p.egn);
 }
 return *this;
}
void People::display() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
People::~People()
{cout << "~People()\n";
 delete name;
 delete egn;
}
class CarPassport: public Car, public People
{public:
  CarPassport(char* = "", unsigned = 0, unsigned = 0,
              char* = "", char* = "");
  ~CarPassport();
  CarPassport(const CarPassport&);
  CarPassport& operator=(const CarPassport&);
  void display() const;
};
CarPassport::CarPassport(char *mark, unsigned year,
           unsigned reg_numb, char* name, char *egn) :
           Car(mark, year, reg_numb), People(name, egn)
{}
CarPassport::~CarPassport()
{cout << "~CarPassport()\n";
}
CarPassport::CarPassport(const CarPassport& cp) : Car(cp),
                                                  People(cp)
{}
CarPassport& CarPassport::operator=(const CarPassport& cp)
{if (this != &cp)
 {Car::operator =(cp);
```

```
  People::operator =(cp);
 }
 return *this;
}
void CarPassport::display() const
{Car::display();
 People::display();
}
void main()
{CarPassport x("FORD FIESTA", 2000, 2295,
               "Vassil Todorov", "8012174586");
 x.display();
 CarPassport y("LADA", 1900, 8817,
               "Sonia Todorova", "8203314576");
 y = x;
 y.display();
}
```

В резултат се получава:

```
Mark: FORD FIESTA
Year: 2000
Reg. Number: 2295
Ime: Vassil Todorov
EGN: 8012174586
Mark: FORD FIESTA
Year: 2000
Reg. Number: 2295
Ime: Vassil Todorov
EGN: 8012174586
~CarPassport()
~People()
~Car()
~CarPassport()
~People()
~Car()
```

Ще отбележим, че в конструктора на производния клас CarPassport тялото е празно, тъй като класът е без собствена член-данна и

единственото му предназначение е обръщение към конструкторите на базовите класове Car и People със съответните им параметри. Тялото на конструктора за присвояване на класа CarPassport също е празно. Присвояването на наследените компоненти се осъществява чрез конструкторите за присояване на основните класове. Това е указано в инициализиращия списък на конструктора за присвояване на CarPassport. Операторната функция за присвояване на CarPassport също се обръща само към операторните функции на основните си класове Car и People. Дефинирането на конструктора за присвояване и на операторната функция за присвояване на производния клас е излишно, тъй като ако бъдат пропуснати, компилаторът ще създаде "служебни", които ще се обърнат към съответните член-функции на основните класове. Същото ще се отнася за конструктора CarPassport(char*, unsigned, unsigned, char*, char*) ако в класа CarPassport не е дефиниран и конструктор за присвояване.

В тази програма е показано също как се осъществява достъпът до наследени компоненти с еднакви имена. И в базовите класове Car и People, и в производния клас CarPassport са дефинирани методи с едно и също име display(). По такъв начин класът CarPassport има три метода с името display() – два наследени от Car и People съответно и един собствен. Проблемът с идентифицирането на имената е разрешен чрез използване на пълните имена, както е показано в дефиницията на метода display() – собствен на класа CarPassport.

**Забележка:** Няма значение как се изброяват основните класове в списъка на производния клас с множествено наследяване. Но вече фиксирана, тази последователност се използва при:

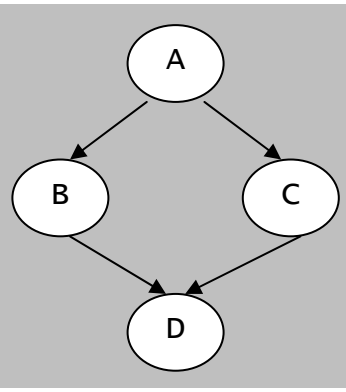
- разполагане на наследените части на производния клас в паметта;
- неявното извикване на конструкторите (деструкторите) на основните класове.

Освен това редът на записване на конструкторите на основния клас в инициализиращия списък на конструктора на производния клас няма значение. Конструкторите ще се активират в реда им в декларацията на производния клас. Ще отбележим още веднъж, че когато се използва множествено наследяване, изискванията към основните класове и съгласуването им с производния клас са както в случая на единичното наследяване.

## 18.2 Виртуални класове

Стандартният механизъм за наследяване дефинира йерархия, която се представя чрез дърво. В този случай, при всяко срещане на основен клас се създава копие на неговите член-данни. Виртуалните основни класове са механизъм за отменяне на стандартния наследствен механизъм.

При реализиране на йерархии на класове с множествено наследяване е възможно един производен клас да наследи многократно даден базов клас. Например, ако класът A има два производни класа B и C, които са базови за класа D (Фиг. 18.3), т.е.

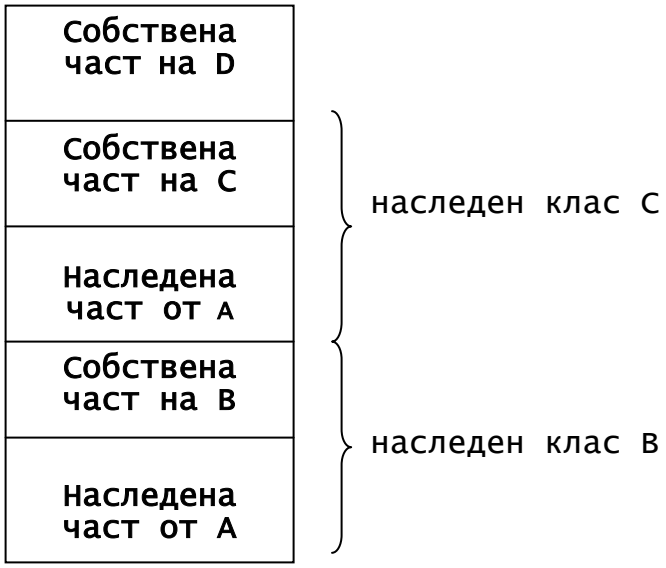



Фиг. 18.3 Една йерархия на класове

```
class A
{...
};
class B: public A
{...
};
class C: public A
{...
};
class D: public B, public C
{...
};
```

Като производни на класа A, класовете B и C наследяват неговите компоненти. От друга страна класовете B и C са базови за класа D. Следователно класът D ще наследи два пъти компонентите на класа A, веднъж чрез класа B и още веднъж – чрез класа на C. За член-функциите двойното наследяване не е от значение, тъй като за всяка член-функция се съхранява само едно копие. Член-променливите обаче се дублират и обект на класа D ще наследи двукратно всяка член-променлива, дефинирана в класа A. Класът D в паметта ще има вида:

class D




Този пример илюстрира един от недостатъците на многократното наследяване на клас – *неефективността от поддържането на множество копия на наследени компоненти*. Ще покажем и други недостатъци на многократното наследяване на класове. Да разгледаме още веднъж горната йерархична схема като попълним декларацията на класовете A, B, C и D. Конструктор по подразбиране няма да дефинираме за нито един от тези класове.

```
class A
{public:
  A(int a)
  {x = a;
  }
  int f() const;
  void print() const;
```

```
  int x;
 };
int A::f() const
{return x;
}
void A::print() const
{cout << "A:: x " << x << endl;
}
class B: public A
{public:
  B(int a, int b): A(a)
  {x = b;
  }
  int f() const;
  void print() const;
  int x;
};
int B::f() const
{return x;
}
void B::print() const
{A::print();
 cout << "B:: x " << x << endl;
}
class C: public A
{public:
  C(int a, int c): A(a)
  {x = c;
  }
  int f() const;
  void print() const;
  int x;
};
int C::f() const
{return x;
}
```

```
void C::print() const
{A::print();
 cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(int a, int b, int c, int d): B(a, b), C(c, d)
  {}
void D::func() const;
void print() const;
};
void D::print() const
{B::print();
 C::print();
}
```

След дефиницията:

```
D d(1, 2, 3, 4);
```

се получава неоднозначност: наследената двукратно член-данна x на класа A има две стойности: 1 и 3. Този проблем можем да разрешим като дефинираме конструктора на класа D по следния начин:

```
D(int a, int b, int c): B(a, b), C(a, c)
  {}
```

но двукратно заделената памет си остава.

Друг проблем възниква при опит в класа D да се използват наследените компоненти A::x, A::f() или A::print(). В случая A::x причината е ясна – има две копия на A::x, но копията на A::f() и A::print() са единствени за класа D. Въпреки това съществува противоречие.

**Пример:** В резултат от компилирането на функцията:

```
void D::func() const
{A::print();
}
```

се получава грешката: *error C2385: 'D::A' is ambiguous*. Причината е, че единственият аргумент на func е this, който е и аргумент на print() в A::print(). this е указател към обект на класа D, който обект съдържа два обекта от основния клас A. Кой от тях да се свърже

с извиканата функция print()? Грижата да се посочи един от двата обекта си остава на програмиста. Ако е необходим само достъп до “конфликтните” наследени членове на клас A (без да се променят стойностите), осъществяването му става чрез последователно прилагане на операцията за явно преобразуване на типове. При атрибут за област public, обект на производен клас може да се преобразува в обект на основен клас с неявни преобразувания, но поради двата клона в йерархичната схема, обект от клас D не може да се преобразува директно в обект от клас A. Възможни са следните последователни преобразувания:

```
D d(1, 2, 3, 4);
(A)(B) d;
(A)(C) d;
```

Като се използват подобни преобразувания, може да се осигури достъп до наследените от класа A членове на класа D. Ще ги илюстрираме с дефиниция на член-функцията func() на D.

```
void D::func() const
{cout << "Derived member x in a part A-B-D "
      << ((A)(B)*this).x << endl
      << "Derived member x in a part A-C-D "
      << ((A)(C)*this).x << endl
      << “Derived member-function f() in a part A-B-D ”
      << ((A)(B)*this).f() << endl
      << “Derived member-function f() in a part A-C-D ”
      << ((A)(C)*this).f() << endl
      << “Derived member-function f() in part C-D”
      << ((C)*this).f() << endl
      << “Derived member-function f() in part B-D”
      << ((B)*this).f() << endl;
}
```

Резултатът от изпълнението на фрагмента:

```
D d(1, 2, 3, 4);
d.func();
```

e:

```
Derived member x in a part A-B-D 1
Derived member x in a part A-C-D 3
```

```
Derived member-function f() in a part A-B-D 1
Derived member-function f() in a part A-C-D 3
Derived member-function f() in part C-D 4
Derived member-function f() in part B-D 2
```

Ще напомним, че указателят this сочи обект от клас D, *this е неговото съдържание, т.е. *this е обект от тип D. Чрез явните преобразувания (A)(B)*this този обект се превръща в обект отначало от клас B, а след това в обект от клас A. Чрез оператора . се прави достъп до член на съответния клас. *При тези преобразувания обектите от класове B и A са* ***временни****. Това е причината, заради която е възможен само достъп, а не промяна на стойности, т.е. при опит за промяна, тя ще е във временна, а не в постоянната памет*.

Ще отбележим също, че обръщението

```
d.print();
```

извиква два пъти метода A::print(), веднъж от обръщението B::print() и друг път от C::print().

**Пример:** Резултатът от изпълнението на фрагмента:

```
D d(1, 2, 3, 4);
d.print();
```

е:

```
A:: x 1
B:: x 2
A:: x 3
C:: x 4
```

Многократното наследяване на клас води от една страна до затруднен достъп до многократно наследените членове, а от друга до поддържане на множество копия на член-данните на многократно наследения клас, което не е ефективно.

Преодоляването на недостатъците на многократното наследяване на клас се осъществява чрез използването на т.н. ***виртуални основни класове***. Чрез тях се дава възможност да се “поделят” основни класове. Когато един клас е виртуален, независимо от участието му в няколко списъка на основни класове, се създава само едно негово копие. В нашия случай, ако класът A се определи като виртуален за класовете B и C, класът D ще съдържа само един “поделен” основен клас A.

**Декларация**

Декларацията на основен клас като виртуален се осъществява като в декларацията на производния клас заедно с името и атрибута за област на основния клас се укаже и запазената дума virtual.

**Пример:** Ще променим декларацията на йерархичната схема от Фиг. 18.3 като определим класа A като виртуален на класовете B и C:

```
class A
{public:
  A(int a)
  {x = a;
  }
  int f() const;
  void print() const;
  int x;
 };
int A::f() const
{return x;
}
void A::print() const
{cout << "A:: x " << x << endl;
}
class B: virtual public A
{public:
  B(int a, int b): A(a)
  {x = b;
  }
  int f() const;
  void print() const;
  int x;
};
int B::f() const
{return x;
}
void B::print() const
{A::print();
```

```
 cout << "B:: x " << x << endl;
}
class C: virtual public A
{public:
  C(int a, int c): A(a)
  {x = c;
  }
  int f() const;
  void print() const;
  int x;
};
int C::f() const
{return x;
}
void C::print() const
{A::print();
 cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(int a, int b, int c, int e): A(a), B(a, b), C(c, e)
  {}
void D::func() const;
void print() const;
};
void D::print() const
{B::print();
 C::print();
}
```

Така класът A е обявен за виртуален. Казва се, че **B и C наследяват класа A виртуално**. Виртуалното наследяване на класа A от класовете B и C оказва влияние само на производните на B и C класове. То не променя поведението на самите класове B и C. Забелязваме, че запазената дума virtual е поставена пред атрибута за област на виртуалния клас A. Всъщност, няма значение редът на virtual и атрибута за област.

Дефинирането и използването на виртуални класове има редица особености. Една от тях касае дефиницията и използването на конструкторите на наследените класове. Нека A е виртуален основен клас за класа B, а класът B е основен за класа D, който пък е основен за класа E. Ако класът A има конструктор с параметри и няма подразбиращ се конструктор, то този конструктор трябва да бъде извикан не само от конструктора на класа B, но и от конструкторите на класовете D и E. Правилото за извикване на конструктори с параметри на виртуални класове може да се изкаже така: конструкторите с параметри на виртуални класове трябва да се извикват от конструкторите на всички класове, които са техни наследници, а не само от конструкторите на преките им наследници. С други думи, производният клас е отговорен за инициализирането на класовете, от които произлиза, както и на всички виртуални основни класове. Ако в инициализиращия списък на конструктора на производен клас няма обръщение към конструктор с параметър на виртуалния клас, използва се неговия подразбиращ се конструктор, ако такъв съществува или се съобщава за отсъстието на подходящ конструктор.

В примера по-горе са дефинирани конструктори с параметри за класовете A, B, C и D. Ще отбележим, че конструкторът с параметри на класа D:

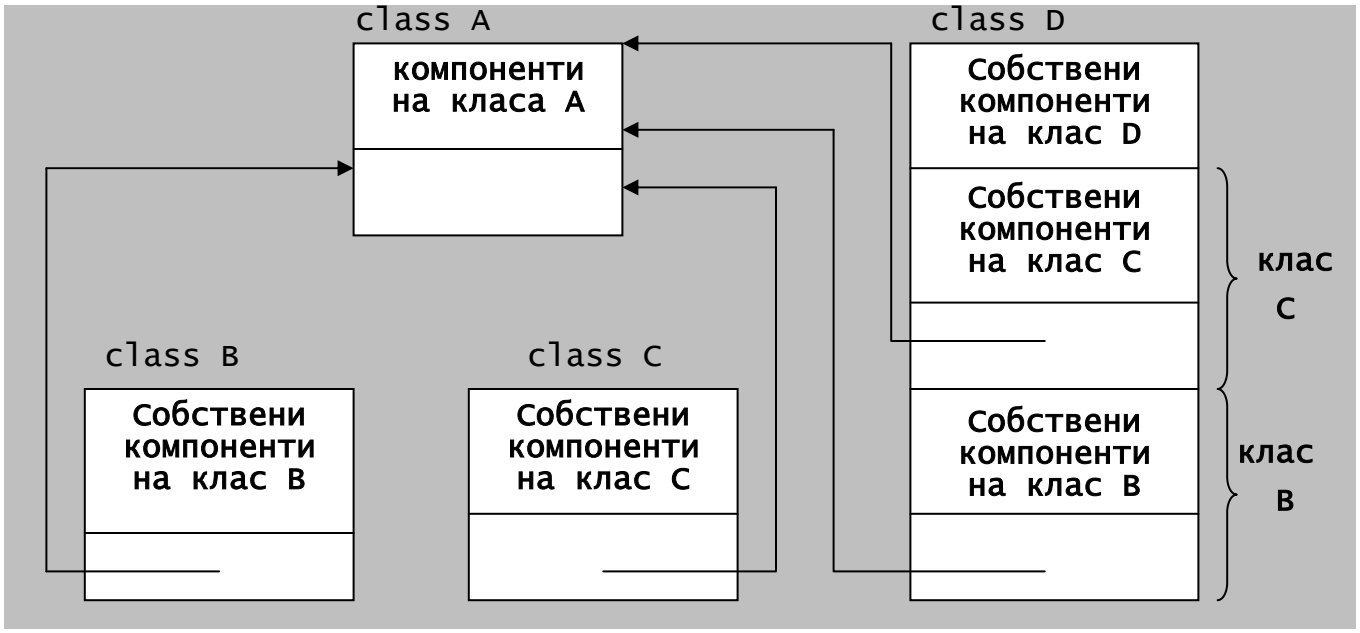
```
D(int a, int b, int c, int d): A(a), B(a, b), C(c, d)
{}
```

явно в инициализиращия си списък се обръща към конструктора на виртуалния основен клас A. Ако това не е направено, компилаторът ще съобщи за грешка, тъй като във виртуалния основен клас A не е определен конструктор по подразбиране. Ако класът A не беше виртуален неговият конструктор нямаше да се изисква в дефиницията на конструктора на класа D и употребата му щеше да доведе до грешка.

Друга особеност е промяната на реда на инициализиране. Инициализирането на виртуални основни класове предхожда инициализирането на другите основни класове в декларацията на производния клас. Ако производен клас наследява основен и виртуален клас, конструкторът на виртуалния клас се извиква първи. При няколко виртуални класа извикването на конструкторите става по реда им в декларацията на производния клас.

*Как използването на виртуални основни класове преодолява недостатъците на многократното наследяване?*

Виртуалният основен клас е общ за всички производни от него класове. Ще се върнем към разглежданата по-горе йерархия на класове (фиг. 18.3) и схематично ще опишем как тя се реализира след обявяването на клас A за виртуален (фиг. 18.4).




Фиг. 18.4 Реализация на виртуално наследяване

Фиг. 18.4 показва как е преодолян проблемът с многократното наследяване на член-данните на клас. Обръщенията A::x, A::f() или A::print() в класа D вече не създават проблем, тъй като класът A се наследява вече само веднъж. Резултатът от изпълнението на фрагмента:

```
   D d(1, 2, 3, 4);
   d.func();
```

е:

```
   Derived member x in a part A-B-D 1
   Derived member x in a part A-C-D 1
   Derived member-function f() in a part A-B-D 1
   Derived member-function f() in a part A-C-D 1
   Derived member-function f() in part C-D 4
   Derived member-function f() in part B-D 2
```

Използването на виртуални основни класове не премахва неедноозначността при достъп до някои член-функции. Например, обръщението

```
d.print();
```

ще извика метода A::print() отново два пъти, т.е. резултатът от изпълнението на обръщението:

```
d.print();
```

е:

```
A:: x 1
B:: x 2
A:: x 1
C:: x 4
```

За да се избегне двукратното изпълнение на A::print() може да се постъпи по следния начин. Дефинициите на методите print() се променят като всеки метод print() се състои от две части: print_own() и print(), използваща наследените print_own() от предшестващите класове.

**Пример:** Примерната програма ще променим до:

```
#include <iostream.h>
class A
{public:
  A(int a)
  {x = a;
  }
 int f() const;
 void print() const
 {print_own();
 }
 private:
  int x;
 protected:
  void print_own() const;
};
int A::f() const
{return x;
}
```

```
void A::print_own() const
{cout << "A:: x " << x << endl;
}
class B: virtual public A
{public:
  B(int a, int b): A(a)
  {x = b;
  }
  int f() const;
  void print() const
  {A::print();
   print_own();
  }
 private:
  int x;
protected:
  void print_own() const;
};
int B::f() const
{return x;
}
void B::print_own() const
{cout << "B:: x " << x << endl;
}
class C: virtual public A
{public:
  C(int a, int c): A(a)
  {x = c;
  }
  int f() const;
  void print() const
  {A::print();
   print_own();
  }
 private:
  int x;
```

```
 protected:
  void print_own() const;
};
int C::f() const
{return x;
}
void C::print_own() const
{cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(int a, int b, int c, int e): A(a), B(a, b), C(c, e)
  {
  }
void print() const
{print_own();
 A::print_own();
 B::print_own();
 C::print_own();
}
protected:
void print_own() const;
};
void D::print_own() const
{
}
void main()
{D d(1,2,3,4);
 d.print();
}
```

Резултат:

```
A:: x 1
B:: x 2
C:: x 4
```

Сега вече е решен и проблемът с двойното изпълнение на член-функцията print() на класа A.

**Задача 166.** Да се дефинира йерархията от фиг. 18.3 като членданните x на класовете A, B, C и D са от тип низ, реализират се в ДП и са капсолирани. За всеки от класовете да се определи “голямата четворка”.

Програма Zad166.cpp решава задачата.

```
// Program Zad166.cpp
#include <iostream.h>
#include <string.h>
class A
{public:
  A(char* = "");
  ~A();
  A(const A&);
  A& operator=(const A &);
  void print() const;
 private:
  char* x;
 };
A::A(char* s)
{x = new char[strlen(s)+1];
 strcpy(x, s);
}
A::~A()
{delete x;
}
A::A(const A& p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
A& A::operator=(const A& p)
{if (this != &p)
{delete x;
 x = new char[strlen(p.x)+1];
```

```
 strcpy(x, p.x);
}
 return *this;
}
void A::print() const
{cout << "A:: x " << x << endl;
}
class B: virtual public A
{public:
  B(char* = "", char* = "");
  ~B();
  B(const B&);
  B& operator=(const B&);
  void print() const;
 private:
  char* x;
};
B::B(char* a, char* b): A(a)
{x = new char[strlen(b)+1];
 strcpy(x, b);
}
B::~B()
{delete x;
}
B::B(const B& p) : A(p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
B& B::operator=(const B& p)
{if (this != &p)
{A::operator=(p);
 delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
```

```
}
void B::print() const
{A::print();
 cout << "B:: x " << x << endl;
}
class C: virtual public A
{public:
  C(char* = "", char* = "");
  ~C();
  C(const C&);
  C& operator=(const C&);
  void print() const;
 private:
  char* x;
};
C::C(char* a, char* b): A(a)
{x = new char[strlen(b)+1];
 strcpy(x, b);
}
C::~C()
{delete x;
}
C::C(const C& p) : A(p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
C& C::operator=(const C& p)
{if (this != &p)
{A::operator=(p);
 delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
}
void C::print() const
```

```
{A::print();
 cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(char* a = "", char* b = "", char* c = "", char* d = "") :
                                  A(a), B(a, b), C(a, c)
  {x = new char[strlen(d)+1];
   strcpy(x, d);
  }
  ~D()
  {cout << "~D(): \n";
   delete x;
  }
  D(const D& p): A(p), B(p), C(p)
  {x = new char[strlen(p.x)+1];
   strcpy(x, p.x);
  }
  D& operator=(const D&p)
  {if(this!=&p)
  {B::operator =(p);
   C::operator =(p);
   delete x;
   x = new char[strlen(p.x)+1];
   strcpy(x, p.x);
  }
  return *this;
  }
 void print() const;
 private:
  char* x;
};
void D::print() const
{B::print();
 C::print();
 cout << "D::x: " << x << endl;
```

```
}
void main()
{D d("Mimi", "Toni", "Liza", "Lora");
 d.print();
 D d1, d2;
 d2 = d1 = d;
 d1.print();
 d2.print();
}
```

Резултат:

Фрагментът

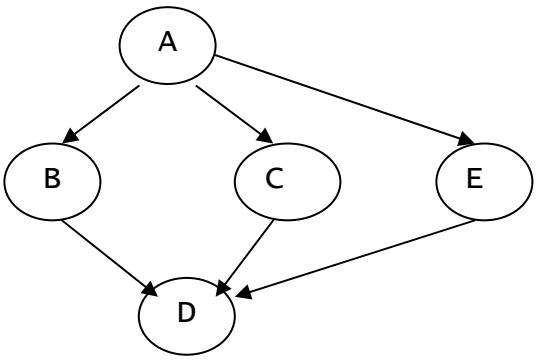
```
A::x Mimi
B::x Toni
A::x Mimi
C::x Liza
D::x Lora
```

се повтаря три пъти, след което също три пъти се извежда:

```
~D():
```

В тази реализация не се грижим за избягване на двойното извеждане на компонентите на класа A.

Характеристиките на виртуалните класове могат да се комбинират с тези на обикновените. Например, може да се разглежда йерархия от вида:




така, че A така, че A е виртуален за класовете B и C и не е виртуален за класа E. Класът D обединява три разклонения: два с виртуален основен клас A и един с наследени членове от обикновения клас A. Механизмът на виртуалните основни класове ще обедини двата виртуални клона в едно виртуално копие на основния клас, но третият клон ще породи още едно наследено копие на основния клас A, използван като

обикновен клас. В този случай се получават нееднозначности при достъп до членовете на двете копия. Преодоляването им става чрез преобразувания.

Нека накрая разгледаме отново познатата йерархична схема от Фиг. 18.3, в която класът A е виртуален за класовете B и C, но атрибутът му за област е public за класа B и private за класа C. Очевидно ситуацията е нееднозначна – обект на D няма достъп до компонентите на класа A по пътя A-C-D, но има такъв по пътя A-B-D. Тази нееднозначност е преодоляна с избора, че ако в някоя декларация виртуалният клас е обявен като public, счита се, че той е с атрибут public навсякъде.

**Пример:** Ако

D d(1, 2, 3, 4);

обръщенията: d.print(); d.A::print(); d.B::print(); d.C::print(); са коректни. А ако

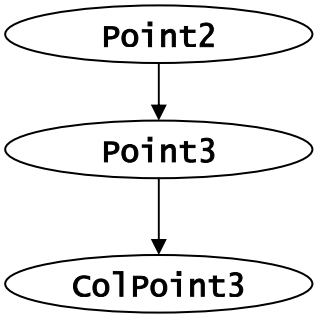
C c(5, 7);

обръщението c.A::print(); не е допустимо заради това, че A е с атрибут за област private в класа C.

## 18.3. Динамично свързване. Виртуални функции

Вече използвахме функции с еднакви имена в т.ч. и методи на класове. В случая на обикновени функции за разпознаването на функцията се използва механизъм, който се изразява в следното: по време на компилация се сравняват формалните с фактическите параметри в обръщението и по правилото за най-доброто съвпадане се избира необходимата функция. След заместване на формалните с фактическите параметри се изпълнява тялото на функцията. При член-функциите на йерархията от класове, конфликтът между имената на наследените и собствените методи от един и същ тип и с едни и същи параметри се разрешава също по време на компилация чрез правилото на локалния приоритет и чрез явно посочване на класа, към който принадлежи методът. В тези два случая тъй като процесът на реализиране на обръщението към функцията приключва по време на компилация и не може да бъде променян по време на изпълнение на програмата се казва, че има **статично разрешаване на връзката** или **статично свързване**.

**Пример:** В следващата програма е дефинирана йерархията:




определяща точка в равнината, точка в тримерното пространство и точка в тримерното пространство с цвят.

```
#include <iostream.h>
class Point2
{public:
 Point2(int a = 0, int o = 0)
 {x = a;
  y = o;
 }
 void Print() const
 {cout << x << ", " << y ;
 }
 private:
  int x, y;
 };
class Point3 : public Point2
{public:
 Point3(int a = 0, int o = 0, int b = 0) : Point2(a, o)
 {z = b;
 }
 void Print() const
 {Point2::Print();
  cout << ", " << z << endl;
 }
 private:
  int z;
};
```

```
class ColPoint3 : public Point3
{public:
 ColPoint3(int a = 0, int o = 0, int b = 0, int c = 0) :
                                        Point3(a, o, b)
 {col = c;
 }
 void Print() const
 {Point3::Print();
  cout << "colour: " << col << endl;
 }
 private:
  int col;
};
void main()
{Point2 p2(5, 10);
 Point3 p3(2, 4, 6);
 ColPoint3 p4(12, 24, 36, 11);
 Point2 *ptr1 = &p3; // атрубутът на Point2 е public
 ptr1->Print();
 cout << endl;
 Point2 *ptr2 = &p4; // атрубутът на Point2 е public
 ptr2->Print();
}
Резултат:
2 4
12 14
```

И в трите класа е дефинирана функция Print() без параметри и от тип void. В главната функция са дефинирани три обекта: p2, p3 и p4 от класове Point2, Point3 и ColPoint3 съответно. Освен това са дефинирани указатели и ptr1 и ptr2 към класа Point2. Указателят ptr1 е инициализиран е с адреса на обекта p3 от класа Point3, а ptr2 – с адреса на обекта p4 от класа ColPoint3. Тъй като атрибутът за област на Point2 в Point3 е public и атрибутът за област на Point3 в ColPoint3 също е public, преобразуванията са допустими. Обръщението: ptr1->Print(); извежда първите две координати на точката p3, а ptr2->Print(); - първите две координати на точката с цвят p4, т.е.

изпълнява се Print() на класа Point2 и в двата случая. Още по време на компилация член-функцията Print() на Point2 е определена като функция на обръщенията ptr1->Print() и ptr2->Print(). Определянето става от типа Point2 на указателите ptr1 и ptr2. Връзката е определена статично и не може да се промени по време на изпълнение на програмата.

Ако искаме след свързването на ptr1 с адреса на p3 да се изпълни член-функцията Print() на Point3, а също след свързването на ptr2 с адреса на p4 да се изпълни член-функцията Print() на ColPoint3 са необходими явни преобразувания от вида:

```
Point2 *ptr1 = &p3;
 ((Point3*)ptr1)->Print();
 cout << endl;
 Point2 *ptr2 = &p4;
 ((ColPoint3*)ptr2)->Print();
```

Отново връзките са разрешени статично.

При статичното свързване по време на създаването на класа трябва да се предвидят възможните обекти, чрез които ще се викат член-функциите му. При сложни йерархии от класове това е не само трудно, но и понякога невъзможно. Езикът C++ поддържа още един механизъм, прилаган върху специален вид член-функции, наречен **късно** или **динамично свързване**. При него изборът на функцията, която трябва да се изпълни, става по време на изпълнение на програмата.

Динамичното свързване капсулира детайлите в реализацията на йерархията. При него не се налага проверка на типа. Текстовете на програмите се опростяват, а промени се налагат много по-рядко. Разширяването на йерархията не създава проблеми. Това обаче е с цената на усложняване на кода и забавяне на процеса на изпълнение на програмата.

Наличието на двата механизма на свързване – статично и динамично, дава възможност на програмиста да се възползва от положителните им страни.

Прилагането на механизма на късното свързване се осъществява върху специални член-функции на класове, наречени **виртуални член-функции** или само **виртуални функции**.

Виртуалните методи се декларират чрез поставяне на запазената дума virtual пред декларацията им, т.е.

```
virtual <тип_на_резултата> <име_на_метод>(<параметри>);
```

**Пример:** В класовете Point2, Point3 и ColPoint3, на програмата от примера по-горе, член-функциите void Print()const са обявени за виртуални.

```
#include <iostream.h>
class Point2
{public:
 Point2(int absc = 0, int ord = 0)
 {x = absc;
  y = ord;
 }
 virtual void Print() const
 {cout << x << ", " << y ;
 }
 private:
  int x, y;
 };
class Point3 : public Point2
{public:
Point3(int a = 0, int o = 0, int b = 0) : Point2(a, o)
{z = b;
}
virtual void Print() const
{Point2::Print();
 cout << ", " << z << endl;
}
private:
  int z;
};
class ColPoint3 : public Point3
{public:
 ColPoint3(int a=0, int o=0, int b=0, int c=0) : Point3(a, o, b)
```

```
 {col = c;
 }
 virtual void Print() const
 {Point3::Print();
  cout << "colour: " << col << endl;
 }
 private:
  int col;
};
void main()
{Point2 p2(5, 10);
 Point3 p3(2, 4, 6);
 ColPoint3 p4(12, 24, 36, 11);
 Point2 *ptr1 = &p3;
 ptr1->Print();
 cout << endl;
 Point2 *ptr2 = &p4;
 ptr2->Print();
}
Резултат:
2, 4, 6
12, 24, 36
colour: 11
```

Декларирането на член-функцията Print() като виртуална причинява обръщенията ptr1->Print(); и ptr2->Print(); да определят функцията, която ще бъде извикана едва при изпълнението на програмата. ***Определянето е в зависимост от типа на обекта, към който сочи указателят, а не от класа към който е указателят.*** В случая, указателят ptr1 е към класа Point2, но сочи обекта p3, който е от класа Point3. Затова обръщението ptr1->Print(); изпълнява Point3::Print(). Указателят ptr2 е към класа Point2, но сочи обекта p4, който е от класа ColPoint3. Затова обръщението ptr2->Print(); изпълнява ColPoint3::Print().

*Ще отбележим, че:*

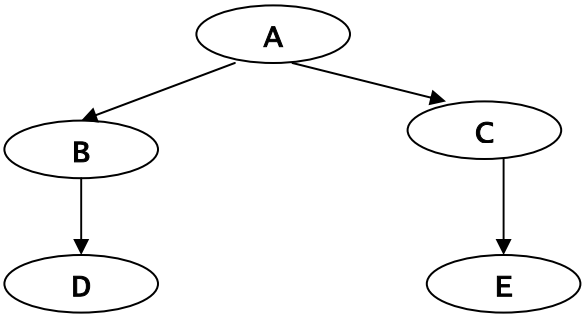
1. Само член-функции на класове могат да се декларират като виртуални. По технически съображения конструкторите не могат да се деларират като виртуални.
2. Ако в даден клас е декларирана виртуална функция, декларираните член-функции със същия прототип (име, параметри и тип на върнатата стойност) в производните на класа класове също са виртуални дори ако запазената дума virtual бъде пропусната.
3. Ако в производен клас е дефинирана функция със същото име като определена вече в основен клас като виртуална член-функция, но с други параметри и/или тип, то това ще е друга функция, която може да бъде или да не бъде декларирана като виртуална.
4. Ако в производен клас е дефинирана виртуална функция със същия прототип като на невиртуална функция на основен клас, то те се интерпретират като различни функции.
5. Възможно е виртуална функция да се дефинира извън клас. Тогава заглавието ѝ не започва със запазената дума virtual, т.е. запазената дума virtual може да се среща само в тялото на клас.
6. Виртуалните функции се наследяват като другите компоненти на класа.
7. Основният клас, в който член-функция е обявена за виртуална, трябва да е с атрибут public в производните от него класове.
8. Виртуалните функции се извикват чрез указател към или псевдоним на обект от някакъв клас.
9. Виртуалната функция, която в действителност се изпълнява, зависи от типа на аргумента.
10. Виртуалните функции не могат да бъдат декларирани като приятели на други класове.

*Някои предимства на виртуалните функции*

1. *Производният клас наследява всяка виртуална функция на базовия клас, за която няма собствена дефиниция*

От тук следва, че не е задължително виртуалните функции да се декларират във всеки клас от йерархията. Ако виртуална функция е дефинирана в базов клас и логиката на производния клас не изисква нейното предефиниране, декларацията й може да се пропусне. Когато бъде извикана виртуална функция за обект от даден клас, тя се търси в него. Ако не е дефинирана в класа, търсенето продължава в базовия клас и нагоре по йерархията.

**Пример:** Нека виртуалната функция void f() е дефинирана като виртуална само в класовете A и B на йерархията:



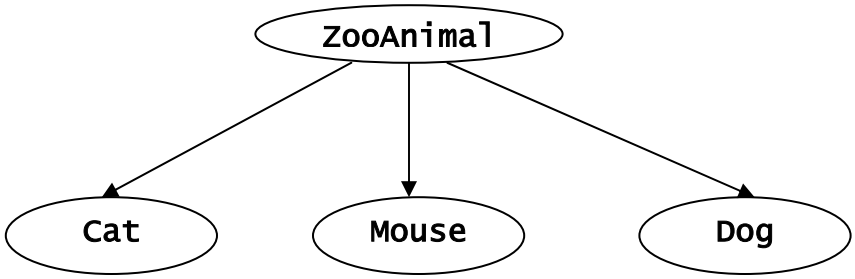

Извикването на функцията f() от обекти от класовете A, C и E ще доведе до изпълнението на функцията A::f(), а нейното извикване за обекти от класовете B и D – ще изпълни функцията B::f(). Ако в класа C бъде дефинирана функция от вида: void f(){}, то функцията A::f() ще се извика само за обект на класа A. За класовете C и E ще бъде извикана празната виртуална функция.

*2. Реализират се полиморфни действия*

**Полиморфизмът** е важна характеристика на ООП. Изразява се в това, че едни и същи действия (в общия смисъл) се реализират по различен начин в зависимост от обектите, върху които се прилагат, т.е. действията са полиморфни (с много форми). Полиморфизмът е свойство на член-функциите на обектите и в езика C++ се реализира чрез виртуални функции. За да се реализира полиморфно действие, класовете върху които то ще се прилага, трябва да имат общ родител или прародител, т.е. да бъдат производни на един и същ клас. В този клас трябва да бъде дефиниран виртуален метод, съответстващ на полиморфното

действие. Във всеки от производните класове този метод може да бъде предефиниран съобразно особеностите на този клас. Активирането полиморфното действие става чрез указател към базовия клас, на който могат да се присвоят адресите на обекти на който и да е от производните класове от йерархията. Ще бъде изпълнен методът на съответния обект, т.е. в зависимост от обекта към който сочи указателят ще бъде изпълняван един или друг метод. Ако класовете, в които трябва да се дефинират виртуални методи нямат общ родител, такъв може да бъде създаден изкуствено чрез дефиниране на т.н. **абстрактен клас**.

**Пример:** В йерархия на класове еднотипни действия са описани с член-функции с еднакви прототипи. Член-функциите на производните класове обикновено извършват редица общи действия. В този случай в основния клас може да се реализира една невиртуална функция, която извършва общите действия и след или преди това извиква виртуалната функция, извършваща специфичните действия на класовете. В следващата програма е дефинирана йерархията от класове:




```
#include <iostream.h>
class ZooAnimal
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Address:\n"
      << "Sofia, Bulgaria\n";
 }
 private:
 //...
 };
class Cat : public ZooAnimal
```

```
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Cat\n";
 }
 //...
};
class Mouse : public ZooAnimal
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Mouse\n";
 }
 //...
};
class Bear : public ZooAnimal
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Bear\n";
 }
 //...
};
void main()
{ZooAnimal zoo; zoo.print();
 Cat c; c.print();
 Mouse m; m.print();
 Bear b; b.print();
 }
```

Резултат:

```
ZooAnimal
Address:
Sofia, Bulgaria
ZooAnimal
Cat
ZooAnimal
```

Mouse
ZooAnimal
Dog

Член-функцията void print()const; на всеки един от класовете извежда общата за всички класове информация:

ZooAnimal

и специфична за всеки клас информация – определяща: адреса на зоологическата градина (в клас ZooAnimal) и вида на животното Cat, Mouse или Dog в производните класове Cat, Mouse или Dog, съответно. Следващата програма е модификация на горната. В класа ZooAnimal е дефинирана обикновена член-функция void print() const, която извежда повтарящия се текст, след което се обръща към виртуалната функция void spec() const;. Тази функция описва специфичните за класовете ZooAnimal, Cat, Mouse и Dog действия. Функцията spec() има един параметър – this. Когато this сочи обект от клас Cat, spec() е функцията Cat::spec(), когато this сочи обект от клас Mouse, spec() е функцията Mouse::spec(), а когато this сочи обект от клас Dog, spec() е функцията Dog::spec().

```
#include <iostream.h>
class ZooAnimal
{public:
 virtual void spec() const
 { cout << "Address:\n"
        << "Sofia, Bulgaria\n";
 }
 void print() const
 {cout << "ZooAnimal\n";
  spec();
 }
 private:
 //...
 };
class Cat : public ZooAnimal
{public:
 virtual void spec() const
 {cout << "Cat\n";
```

```
 }
 //...
};
class Mouse : public ZooAnimal
{public:
virtual void spec() const
 {cout << "Mouse\n";
 }
 //...
};
class Bear : public ZooAnimal
{public:
virtual void spec() const
 {cout << "Bear\n";
 }
 //...
};
void main()
{ZooAnimal zoo; zoo.print();
 Cat c; c.print();
 Mouse m; m.print();
 Bear b; b.print();
}
```

В случая, общият повтарящ се код е малък по обем, но има йерархии, където това не е така.

Същият резултат се получава след изпълнение на фрагмента:

```
ZooAnimal zoo, *pzoo;
Cat c; Mouse m; Bear b;
pzoo = &zoo; pzoo->print();
pzoo = &c; pzoo->print();
pzoo = &m; pzoo->print();
pzoo = &b; pzoo->print();
```

Забелязваме, че едно и също обръщение: pzoo->print(); е извикано четири пъти и всеки път изпълнява член-функцията print() с различни обръщения към виртуалната функция spec(). Обръщението pzoo->print() се разрешава статично, тъй като print() не е виртуална. Полиморфният

й характер произлиза от съдържащата се в нея виртуална вункция spec().

Преди да разгледаме абстрактните класове, ще се спрем на още един важен въпрос – **достъпът до виртуална функция**. Всяка член-функция на клас, в който е дефинирана виртуална функция, има пряк достъп до виртуалната функция, т.е. на локално ниво достъпът се определя по традиционните правила. На глобално ниво достъпът е малко по-различен. Преди да изкажем правилото, ще разгледаме следната примерна програма:

```
#include <iostream.h>
class Base
{public:
 virtual void pub()
 {cout << "pub()\n";
  //....
 }
 void usual()
 {cout << "usual()\n";
  pub();
  pri();
  pro();
 }
private:
 virtual void pri()
 {cout << "pri()\n";
  //....
 }
 protected:
 virtual void pro()
 {cout << "pro()\n";
  //....
 }
};
class Der : public Base
{protected:
 virtual void pub()
 {cout << "Derived class\n";
```

```
     Base::pub();
     Base::pro();
    }
    public:
    virtual void pri()
    {cout << "Derived-pri()\n";
    }
    virtual void pro()
    {cout << "Derived-pro()\n";
    }
   };
   void main()
   {Base *p = new Base;
    Base *q = new Der;
    p->pub();
    q->pub();
   // p->pri();
   // q->pri();
    Der *r = new Der;
    r->pri();
   // q->pro();
   // r->pub();
    p->usual();
   }
```

В нея е реализирана йерархията Base -> Der, като в класа Base са дефинирани три виртуални функции:

void pub(); - в секция public

void pri(); - в секция private

void pro(); - в секция protected

и една обикновена член-функция:

void usual(); - в секция public, която ги използва.

В класа Der са предефинирани трите виртуални функции, но с променен достъп:

void pub(); - в секция protected

void pri(); и void pro(); - в секция public.

В главната функция са дефинирани два указателя p и q към класа Base и указател r към производния клас Der. Тъй като функцията pub() е виртуална, десните страни на дефинициите:

```
Base *p = new Base;
Base *q = new Der;
```

определят, че в обръщенията:

```
p->pub();
q->pub();
```

p ще активира Base::pub(), а q – Der::pub(), ако е възможен достъп. Достъпът се определя от вида на секцията на метода pub() в класовете към които сочат указателите. Тъй като и p, и q са от тип Base* (т.е. сочат към клас Base) и в Base pub() е в секция public, независимо, че Der::pub() е в секция protected, обръщенията се изпълняват.

Обръщенията:

```
// p->pri();
// q->pri();
```

са коментирани, тъй като не са успешни. Член-функцията pri() е виртуална и тъй като p и q не са променени, десните страни на дефинициите:

```
Base *p = new Base;
Base *q = new Der;
```

определят, че p ще активира Base::pri();, а q – Der::pri(), ако е възможен достъп. Достъпът се определя от вида на секцията, в която се намира pri() в класа Base (към него сочат и p, и q). Тъй като секцията е private, достъпът е невъзможен.

Дефиницията

```
Der *r = new Der;
```

определя r като указател към Der и го свързва с обект от него. Функцията pri() е виртуална. Според дясната страна на дефиницията на r, обръщението:

```
r->pri();
```

активира Der::pri(). Тъй като r е указател към Der, а в класа Der функцията pri() е дефинирана в секция public, достъпът е възможен.

Обръщенията:

```
// q->pro();
// r->pub();
```

отново са коментирани, тъй като не са успешни. В първия случай виртуалната функция pro() е дефинирана в секция protected в класа Base, към който сочи указателят q. Във втория случай виртуалната функция pub() е дефинирана в секция protected в класа Der, към който сочи указателят r.

Обръщението

```
p->usual();
```

е допустимо, тъй като обикновената член-функция на класа Base е дефинирана в секция public.

Ще заключим, че достъпът до виртуална функция на глобално ниво зависи от секцията, в която е дефинирана тя, на класа към който сочи указателят, чрез който се активира функцията.

*Съществуват три случая, при които обръщението към виртуална функция се решава статично (по време на компилация):*

1. *Виртуалната функция се извиква чрез обект на класа, в който е дефинирана*

**Пример:** Фрагментът

```
Cat c; c.spec();
Mouse m; m.spec();
Bear b; b.spec();
```

е допустим. Независимо че spec() е виртуална, предварително (по време на компилация) се определя, че в c.spec() се извиква spec() на класа Cat, че в m.spec() се извиква spec() на класа Mouse и че в b.spec() се извиква spec() на класа Bear. Ще отбележим изрично, че методите void spec() са обявени в секция public. В противен случай достъпът е невъзможен.

Ще отбележим също, че ако

```
ZooAnimal *z;
```

обръщението:

```
(*z).spec();
```

е виртуално.

2. *Виртуалната функция се активира чрез указател към или псевдоним на обект, но явно, чрез операцията ::, е посочена конкретната функция*

**Пример:**

```
ZooAnimal *pz;
```

```
Bear b; Cat c; Mouse m;
pz = &b; pz -> spec(); // динамично свързване
pz = &c; pz -> spec(); // динамично свързване
pz = &m; pz -> spec(); // динамично свързване
pz->ZooAnimal::spec();  // статично свързване
```

Отново ще отбележим, че методът void spec() е в секция public за всеки от класовете на йерархията с основен клас ZooAnimal.

*3. Виртуалната функция се активира с тялото на конструктор или деструктор на основен клас.*

Това е така, защото обектът от производния клас още не е създаден или вече е разрушен.

## 18. 4. Абстрактни класове. Контейнерни класове

Възможно е виртуалните функции да имат само декларация, а не дефиниция. Такива виртуални член-функции се наричат **чисти**. За да се определи една виртуална функция като чиста се използва следният синтаксис:

```
virtual <тип> <име_на_функция>(<параметри>) = 0;
```

Клас, в който е декларирана поне една чиста виртуална функция се нарича **абстрактен**. Абстрактните класове се характеризират със следните свойства:

а) Обекти от тези класове не могат да се създават;

б) Чистите виртуални функции, задължително трябва да бъдат предефинирани в производните класове със същите прототипи или да бъдат овявени за чисто виртуални в тях. В последния случай, класът наследник също е абстрактен.

Абстрактните класове са предназначени да служат като базови на други класове. Чрез тях се обединяват в обща структура различни йерархии.

Полиморфизмът позволява създаването на класове с различна логическа структура, които могат да включват обекти от други класове. Логическата структура на класа се реализира отделно от обектите, които се включват в него. Връзката между тях се реализира чрез указатели към контейнери, които съхраняват обекти от различни класове.

Логическата структура на контейнерните класове може да е най-различна – масив, списък, множество и т.н. Ще предложим реализацията на едносвързан списък, контейнерите на който могат да съхраняват обекти от два класа: Point2 и Point3.

**Задача 167.** Да се напише програма, която създава хетерогенен едносвързан списък, контейнерите на който могат да съхраняват обекти от два класа: Point2 и Point3.

```
// Program Zad167.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<void*> lst;
class Object
{public:
 virtual void Print()=0;
};
class lst_het: public lst, public Object
{public:
 lst_het(){}
 void Print();
};
void lst_het::Print()
{IterStart();
 elem<void*> *p = Iter();
 Object *ptr;
 while(p)
 {ptr = (Object*)(p->inf);
  ptr->Print();
  p = p->link;
 }
 cout << endl;
}
class Point2:Object
{public:
 Point2(int absc = 0, int ord = 0)
```

```
 {x = absc;
  y = ord;
 }
 void Print()
 {cout << x << ", " << y << endl;
 }
 private:
  int x, y;
 };
class Point3 : Object
{public:
 Point3(int a, int o, int b)
 {x = a;
  y = o;
  z = b;
 }
 void Print()
 {cout << x << ", " << y << ", " << z << endl;
 }
 private:
  int x, y, z;
};
void main()
{lst_het lh;
 Point2 p21(1,5), p22(2,6);
 Point3 p31(10, 20, 30), p32(11, 21, 31);
 lh.ToEnd(&p21); lh.ToEnd(&p31);
 lh.ToEnd(&p22); lh.ToEnd(&p32);
 lh.Print();
}
```

(Виртуални деструктори)

## Задачи

## Допълнителна литература

1 B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.

2. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.

## Упражнение 3-1

## Наследяване. Производни класове – дефиниране, достъп

**Забележка:** В това упражнение да не се използват конструктори, деструктори, операторна функция за присвояване и конструктор за присвояване.

**Задача 1.** Какъв е резултатът от изпълнението на програмата:

```
#include <iostream.h>
class A
{public:
 void print(int n) const
 {cout << n << endl;
 }
 };
class D : public A
{public:
 void printD(int n) const
 {if (n<=1) print(n);
  else if (n%2 ==0) printD(n/2);
  else printD(3*n+1);
 }
};
int main()
{D d;
 d.printD(3);
 return 0;
}
```

Определете отговора без да компилирате и изпълнявате програмата.

Отговорът е 1.

Ще се промени ли поведението на програмата ако атрибутът за област в наследения клас D се промени от public в private или protected? (NE)

Променете програмата като използвате един и същ идентификатор за извеждане print, т.е.

```
#include <iostream.h>
class A
{public:
 void print(int n) const
 {cout << n << endl;
 }
 };
class D : public A
{public:
 void print(int n) const
 {if (n<=1) A::print(n);    // пълно име
  else if (n%2 ==0) print(n/2);
  else print(3*n+1);
 }
};
int main()
{D d;
 d.print(3);
 return 0;
}
```

**Изкажете правилото за приоритета на локалното име (вижте лекцията).**

**Задача 2.** Да се дефинира клас “Точка в равнината”. Като се използва този клас, да се определи клас “Точка в тримерното пространство”. И най-накрая да се определи класът “Точка в тримерното пространство с цвят”, наследник на клас “Точка в тримерното пространство”. Цвят се задава чрез цяло число.

Една примерна програма е:

```
#include <iostream.h>
class Point2
{public:
 void ReadPoint2(int absc = 0, int ord = 0)
```

```
 {x = absc;
  y = ord;
 }
 void PrintPoint2()
 {cout << x << ", " << y ;
 }
 private:
  int x, y;
 };
class Point3 : public Point2
{public:
 void ReadPoint3(int a, int o, int b)
 {ReadPoint2(a, o);
  z = b;
 }
 void PrintPoint3()
 {PrintPoint2();
  cout << ", " << z << endl;
 }
 private:
  int z;
};
class ColPoint3 : public Point3
{public:
 void ReadColPoint3(int a, int o, int b, int c)
 {ReadPoint3(a, o, b);
  col = c;
 }
 void PrintColPoint3()
 {PrintPoint3();
  cout << "collor: " << col << endl;
 }
 private:
  int col;
};
```

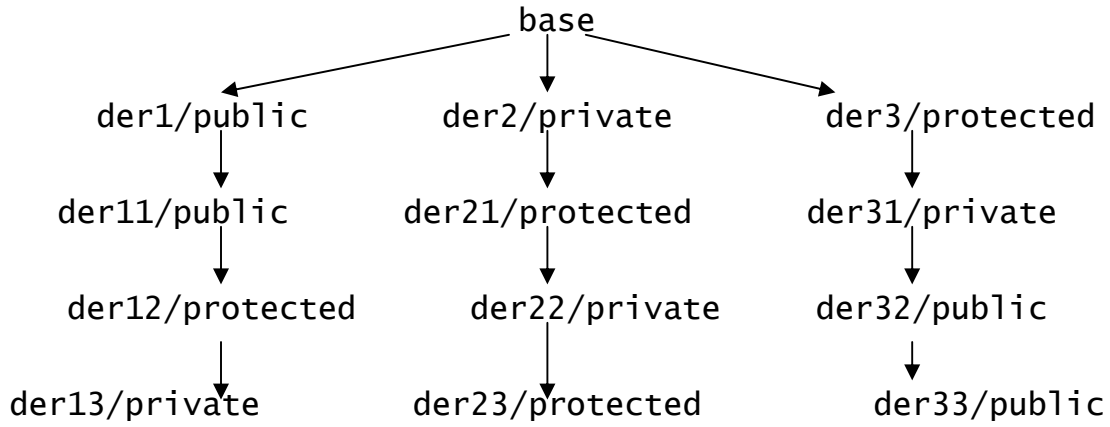
```
int main()
{Point2 p2;
 p2.ReadPoint2(5,10);
 p2.PrintPoint2();
 cout << endl;
 Point3 p3;
 p3.ReadPoint3(2,4,6);
 p3.PrintPoint3();
 ColPoint3 p4;
 p4.ReadColPoint3(2,4,6, 111);
 p4.PrintColPoint3();
 return 0;
}
```

Променете атрибутите за област и определете:

а) вида на наследяване на компонентите;

б) достъпа на член-функциите и на p1, p2, p3, p4 до компонентите на базовите класове.

**Задача 3.** Дефинирайте следната йерархия:




и определете:

а) вида на наследяване на компонентите;

б) достъпа на член-функциите и на обектите и външните функции до компонентите на базовите класове.

(оставете ги сами да работят, след което напишете част от йерархията)

Други задачи.

## Упражнение 3-2
## Наследяване. Производни класове – конструктори, деструктори

**Задача 1**. Какъв е резултатът от изпълнението на програмата?

```
#include <iostream.h>
class base
{public:
 void init(int x)
 {bx = x;
 }
 void display() const
 {cout << " class base: bx= " << bx << endl;
 }
 protected:
  int bx;
 private:
// …
};
class der: public base
{public:
 void init(int x)
 {bx = x;
  base::bx = x + 5;
 }
 void display() const
 {cout << " class der: bx = " << bx;
  cout << " base::bx = " << base::bx << endl;
 }
 protected:
  int bx;
 private:
 //...
};
```

```
class derder: public der
{public:
 void init(int x)
 {bx = x;
  base::bx = x + 5;
  der::bx = x + 10;
 }
 void display() const
 {cout << " class der-der: bx = " << bx;
 cout << " class der: bx = " << der::bx;
  cout << " base::bx = " << base::bx << endl;
 }
 protected:
  int bx;
 private:
 //...
};
void main()
{base b;
 der d;
 derder dd;
 b.init(5); d.init(10); dd.init(100);
 b.display(); d.display(); dd.display();
 d.base::init(20);
 d.base::display();
 d.display();
 b.display();
}
```

**Задача 2.** Какъв ще е резултатът от изпълнението на програмата:

```
#include <iostream.h>
class B
{public:
   B();
   B(int n);
};
```

```
   B::B()
   {cout << “B::B()\n”;
   }
   B::B(int n)
   {cout << “B::B(“ << n << “)\n”;
   }
   class D : public B
   {public:
    D();
    D(int n);
    private: B b;
   };
   D::D()
   {cout << “D::D()\n”;
   }
   D::D(int n) : B(n)
   {b = B(-n);
    cout << “D::D(“ << n << “)\n”;
   }
 int main()
 {D d(3);
 return 0;
 }
```

Определете отговора на ръка – без да използвате компютър.

Отг.

```
   B::B(3)
   B::B()
   B::B(-3)
   D::D(3)
```

Обяснете резултата от втората линия (заради наличието на декларацията B b;, определяща обект). След това променете програмата в:

```
   #include <iostream.h>
   class B
   {public:
      B();
```

```
   B(int n);
};
B::B()
{cout << "B::B()\n";
}
B::B(int n)
{cout << "B::B(" << n << ")\n";
}
class D : public B
{public:
 D();
 D(int n);
 private: B b;
          B c;    // нов фрагмент
};
D::D()
{cout << "D::D()\n";
}
D::D(int n) : B(n)
{b = B(-n);
 cout << "D::D(" << n << ")\n";
}
 int main()
 {D d(3);
 return 0;
 }
```

**Рез:** два пъти ще се извика B() – за обекта b и за обекта c.

**Задача 3.** Намерете грешките в дефинициите на класовете:

```
class B
{public:
 B();
 B(int n);
 void print() const;
 private:
 int b;
};
```

```
B::B()
{b=0;
}
B::B(int n)
{b = n;
}
void B:: print() const
{cout << “B: “ << b << endl;
}
class D : public B
{public:
 D();
 D(int n);
 void print(int n) const;
 private:
 B b;
};
D::D()
{}
D::D(int n) : B(n)
{b = n;
}
D::print()const
{cout << “D: “ << b << endl;
}
```

Поправете ги!!! Naprimer:

```
#include <iostream.h>
class B
{public:
 B();
 B(int n);
void print() const;
private:
 int b;
};
B::B()
```

```
{b=0;
}
B::B(int n)
{b = n;
}
void B:: print() const
{cout << "B: " << b << endl;
}
class D : public B
{public:
 D();
 D(int n);
 void print() const;
 private:
 B b;
};
D::D()
{}
D::D(int n) : B(n)
{b = n;
}
void D::print()const
{cout << "D: " ;
 b.print();
}
 int main()
 {D d(3);
  d.print();
 return 0;
 }
```

## Упражнение 3

## Наследяване. Дефиниране на конструктори за присвояване и операторни функции за присвояване на производни класове

**Задача 1.** Изпълнете стъпка по стъпка програмата:

```
#include <iostream.h>
class base
{public:
 base(int x)
 {b = x;}
 base(const base &x)
 {b = x.b + 1;
 }
 protected:
  int b;
};
class der1 : public base
{public:
 der1(int x = 1) : base(x)
 {d = x;
 }
 der1(const der1& x): base(x) // кой base се използва?
 {d = x.d + 2;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
void main()
{der1 d11(5);
 der1 d12 = d11;
 cout << "d11: "; d11.Print();
 cout << "d12: "; d12.Print();
}
```

Възможно ли е конструкторът за присвояване да се промени в:

```
der1(const der1& x): base(x+5)
 {d = x.d + 2;
 }
```

(не, защото + не е предефинирано за x от der1)

**Задача 2.** Какъв е резултатът от изпълнението на програмата:

```
#include <iostream.h>
class base
{public:
 base(int x)
 {b = x;}
 base& operator=(const base &x)
 {b = x.b + 1;
  return *this;
 }
 protected:
  int b;
};
class der1 : public base
{public:
der1(int x = 1) :base(x)
 {d = x;
 }
 der1& operator=(const der1& x)
 {d = x.d + 2;
  b = x.b + 3;
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
class der2 : public base
{public:
```

```
der2(int x = 2) : base(x)
 {d = x;
 }
 der2& operator=(const der2& x)
 {d = x.d + 3;
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
class der3 : public base
{public:
der3(int x = 3) : base(x)
 {d = x;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
void main()
{der1 d11(5), d12;
 der2 d21(5), d22;
 der3 d31(5), d32;
 d12 = d11;
 d22 = d21;
 d32 = d31;
 cout << "d11: "; d11.Print();
 cout << "d12: "; d12.Print();
 cout << "d21: "; d21.Print();
 cout << "d22: "; d22.Print();
 cout << "d31: "; d31.Print();
```

```
 cout << "d32: "; d32.Print();
}
```

рез:

```
d11: der: 5  base: 5
d12: der: 7  base: 8
d21: der: 5  base: 5
d22: der: 8  base: 2
d31: der: 5  base: 5
d32: der: 5  base: 6
```

Изпълнете стъпка по стъпка.

**Задача 3.** Като задача 2, малки промени има. Пак я изпълнете стъпка по стъпка. (Използвайте текста на първата задача, който сте написали вече на дъската)

```
#include <iostream.h>
class base
{public:
 protected:
  int b;
};
class der1 : public base
{public:
der1(int x = 1)
 {d = x;
 }
 der1& operator=(const der1& x)
 {d = x.d + 2;
  b = x.b + 3;
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
```

```
};
class der2 : public base
{public:
 der2(int x = 2)
 {d = x;
 }
 der2& operator=(const der2& x)
 {d = x.d + 3;
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
class der3 : public base
{public:
 der3(int x = 3)
 {d = x;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
void main()
{der1 d11(5), d12;
 der2 d21(5), d22;
 der3 d31(5), d32;
 d12 = d11;
 d22 = d21;
 d32 = d31;
 cout << "d11: "; d11.Print();
 cout << "d12: "; d12.Print();
```

```
 cout << "d21: "; d21.Print();
 cout << "d22: "; d22.Print();
 cout << "d31: "; d31.Print();
 cout << "d32: "; d32.Print();
}
```

Направете други (полезни според вас) промени и изпълнете стъпка по стъпка.

**Задача 4.** Напишете основен клас Worker (работник), който определя работник с име и заплащане за 1 час. Напишете два производни на Worker класа HourlyWorler (почасов работник) и SalariedWorker (щатен работник). За всеки вид работник са дадени: броят на часовете, които той е работил през седмицата; видът работа, която е извършвал (низ) /само един вид работа е работил/. Почасовият работник получава заплата за седмица по следното правило: Всеки час до 40 часа се заплаща по указаната цена. За всеки час от 41 до 60 се заплаща 1.5 пъти повече от указаната цена и за часовете на 60 – 2 пъти повече. Щатният работник получава заплата за 40 часа независомо колко е работил. Класовете да реализират голямата четворка /конструктор по подразбиране, деструктор, констр. за присвояване и операторна функция за присвояване/. Да се създадат масиви, съдържащи двата вида работници. Да се пресметне и изведе заплатата на всеки работник. Да се изведат работниците от всеки вид, сортирани по заплата.

Оставяте ги да работят самостоятелно. На лаборат упражн. да я изпълнят. Вижте решените задачи в лекциите. Преобразуване от вида:

```
     (Student)(*this) = (Student)st;
```

не се извършва във всяка реализация. Затова е заменено с Student::operator=(st).

Дайте им и други задачи.

## Упражнение 4

## Преобразувания на типове. Шаблони на наследени класове

**Задача 1.** Има ли грешки в програмата? Ако не, какъв е резултатът от изпълнението й?

```
#include <iostream.h>
class base
{public:
 base(int x = 0)
 {b = x;
 }
 int get_b() const
 {return b;
 }
 void f()const
 {cout << "b: " << b << endl;
 }
 void f1()const
 {cout << "f1:\n";
 }
 private:
  int b;
 };
class der : public base
{public:
 der(int x = 0) : base(x)
 {d = 5;
 }
 int get_d() const
 {return d;
 }
 void f_der()const
 {cout << "class der: d: " << d
           << " b: " << get_b() << endl;
 }
 private:
  int d;
```

```
 };
void main()
{void (base::*pb)()const = base::f;
 void (der::*pd)()const = pb;
 der y(20);
 (y.*pd)();
}
```

**Задача 2.** Нека класът D е наследник на класа B. Кои от следните присвоявания са допустими?

```
B b;
D d;
B* pb;
D* pd;
b = d;
d = b;
pd = pb;
pb = pd;
d = pd;
b = *pd;
*pd = *pb;
```

**Задача 3.** Дефинирайте шаблон на клас Point3, определящ точка в тримерното пространство с координати от тип T. Определете шаблон на производен клас ColPoint3 на класа Point, определящ точка в тримерното пространство с координати от тип T и цвят от тип U. Дефинирайте и шаблон на клас, определящ точка в тримерното пространство с координати от тип T, цвят от тип U и тегло V. Дефинирайте масив от точки в тримерното пространство с цвят и тегло.

а) Намерете най-тежката точка от масива с цвят в диапазона [1, 5] и лежаща в равнината ax+by+cz = d (a, b, c, d са дадени реални числа).

б) Сортирайте в низходящ ред по тегло точките с цвят от диапазона [1, 10], лежащи в кълбото с център координатното начало и радиус R.

Други задачи.

## Упражнение 5 и 6
## Множествено наследяване. Виртуални класове и функции

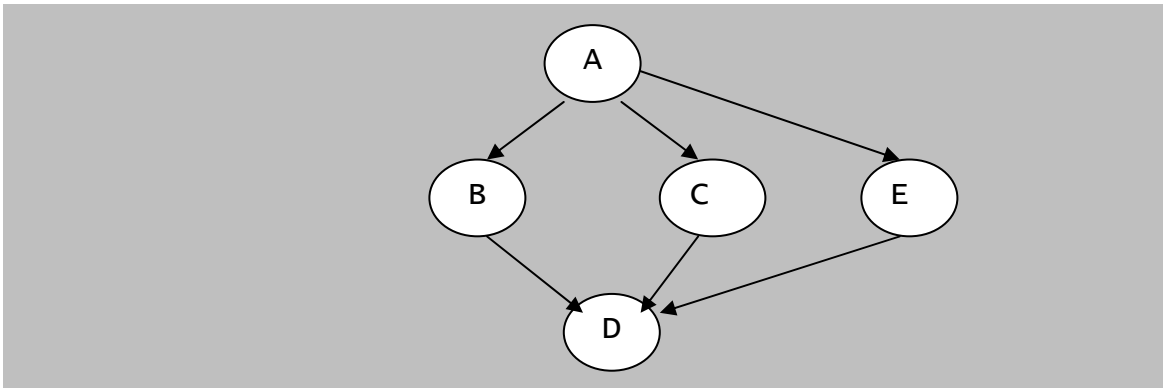
Задача 1. Какъв резултат ще изведе следната програма:

```
#include <iostream.h>
class A
{public:
 A(int a = 1)
{n = a; x = 1.1;
 cout << "A: " << n << "," << x << endl;
}
 private:
 int n;
 double x;
};
class B
{public:
 B(double b = 1)
{n = 2; y = b;
 cout << "B: " << n << "," << y << endl;
}
 private:
 int n;
 double y;
};
class C : public B, public A
{public:
C(int x = 1, int y = 2, int z = 3, double u = 0.0):
A(x), B(y)
{n = z; m = x*y;
 cout << "C: " << n << "," << m << endl;
}
 private:
 int n, m;
};
```

```
void main()
{C c1;
 C c2(2,4,6,1.0);
}
```

Нарисувайте разположението на обектите c1 и c2 в ОП. Ще се промени ли резултатът ако се променят атрибутите за област.

**Задача 2.** Да се изгради йерархията:




така, че A е виртуален за класовете B и C и не е виртуален за класа E. Класовете да съдържат голямата четворка.

(не съм гледала решението по-долу, взех го от стар файл, прожерете го)

```
#include <iostream.h>
#include <string.h>
class A
{public:
  A(char* = "");
  ~A();
  A(const A&);
  A& operator=(const A &);
  void print() const;
 private:
  char* x;
 };
A::A(char* s)
{x = new char[strlen(s)+1];
 strcpy(x, s);
}
```

```
A::~A()
{delete x;
}
A::A(const A& p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
A& A::operator=(const A& p)
{if (this != &p)
{delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
}
void A::print() const
{cout << "A:: x " << x << endl;
}
class B: virtual public A
{public:
  B(char* = "", char* = "");
  ~B();
  B(const B&);
  B& operator=(const B&);
  void print() const;
 private:
  char* x;
};
B::B(char* a, char* b): A(a)
{x = new char[strlen(b)+1];
 strcpy(x, b);
}
B::~B()
{delete x;
}
B::B(const B& p) : A(p)
```

```
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
B& B::operator=(const B& p)
{if (this != &p)
{A::operator=(p);
 delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
}
void B::print() const
{A::print();
 cout << "B:: x " << x << endl;
}
class C: virtual public A
{public:
  C(char* = "", char* = "");
  ~C();
  C(const C&);
  C& operator=(const C&);
  void print() const;
 private:
  char* x;
};
C::C(char* a, char* b): A(a)
{x = new char[strlen(b)+1];
 strcpy(x, b);
}
C::~C()
{delete x;
}
C::C(const C& p) : A(p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
```

```
}
C& C::operator=(const C& p)
{if (this != &p)
{A::operator=(p);
 delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
}
void C::print() const
{A::print();
 cout << "C:: x " << x << endl;
}
class E: public A
{public:
  E(char* = "", char* = "");
  ~E();
  E(const E&);
  E& operator=(const E&);
  void print() const;
 private:
  char* x;
};
E::E(char* a, char* b): A(a)
{x = new char[strlen(b)+1];
 strcpy(x, b);
}
E::~E()
{delete x;
}
E::E(const E& p) : A(p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
E& E::operator=(const E& p)
```

```
{if (this != &p)
{A::operator=(p);
 delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
}
void E::print() const
{A::print();
 cout << "E:: x " << x << endl;
}
class D: public B, public C, public E
{public:
  D(char* a = "", char* b = "", char* c = "",char* d = "",
     char* e = "") : A(a), B(a, b), C(a, c), E(a, d)
  {x = new char[strlen(e)+1];
   strcpy(x, e);
  }
  ~D()
  {cout << "~D(): \n";
   delete x;
  }
  D(const D& p):  B(p), C(p), E(p)
  {x = new char[strlen(p.x)+1];
   strcpy(x, p.x);
  }
  D& operator=(const D&p)
  {if(this!=&p)
  {B::operator =(p);
   C::operator =(p);
   E::operator =(p);
   delete x;
   x = new char[strlen(p.x)+1];
   strcpy(x, p.x);
  }
```

```
  return *this;
  }
 void print() const;
 private:
  char* x;
};
void D::print() const
{B::print();
 C::print();
 E::print();
 cout << "D::x: " << x << endl;
}
void main()
{D d("Mimi", "Toni", "Liza", "Lora", "Vesi");
 d.print();
 D d1, d2;
 d1 = d2 = d;
 d1.print();
 d2.print();
}
```

Коментирайте конструкторите на клас D. Този с 5 параметъра използва обръщение до конструктора на виртуалния клас A, а констр. за присвояване на D, не (не може да направи преобр. A(p)) и използва констр. по подразбиране. Променете конструктора за присвояване на класа D като използвате явно преобразуване:

```
D(const D& p): A((A)(B)p), B(p), C(p), E(p)
  {x = new char[strlen(p.x)+1];
   strcpy(x, p.x);
  }
```

или по другия клон /A((A)(C)p)/.

---

**Задача 3.** Коментирайте програмата:

```
#include <iostream.h>
class Base
```

```
{virtual void f()
 {cout << "Base - f()\n";
 }
 virtual void g()
 {cout << "Base - g()\n";
 }
 void h()
 {cout << "Base - h()\n";
 }
};
class Der : public Base
{public:
 virtual void f()
 {cout << "Der - f()\n";
 }
void g(int x)
 {int y;
  y = x;
  cout << "y= " << y << endl;
 }
 void h()
 {cout << "Der - h()\n";
 }
};
void main()
{int a; Der x; Base y;
 Base *bp = &x;
 bp->f();
 bp->g();
 bp->h();
 bp = &y;
 bp->f();
}
```

Кои обръщения не са коректни? Защо?

Задача 4. Кои от следващите извиквания са статично свързани и кои динамично? Какво извежда програмата?

```
#include <iostream.h>
class B
{public:
 B(){}
 virtual void p() const
 {cout << “B::p\n”;
 }
 void q() const
 {cout << “B::q\n”;
 }

};
class D : public B
{public:
 D(){}
 void p() const
 {cout << “D::p\n”;
 }
 void q() const
 {cout << “D::q\n”;
 }
};
int main()
{B b; D d;
B* pb = new B;
B* pd = new D;
B* pd2 = new D;
b.p(); b.q();
d.p(); d.q();
pb->p(); pb->q();
pd->p(); pd->q();
pd2->p(); pd2->q();
return 0;
```

}

измислете и други задачи