ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/278033468

Основи на програмирането със С++

Book · January 2014

DOI: 10.13140/RG.2.1.5074.6320

CITATIONS
0

READS
8,666

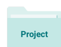
1 author:

Emiliyan Petkov
Veliko Tarnovo University

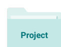
26 PUBLICATIONS 27 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



System training and simulations with augmented reality. View project



ANALYSIS OF TECHNOLOGIES, APPROACHES AND METHODOLOGIES FOR 3D TRAINING OF STUDENTS IN STEREOMETRY View project

All content following this page was uploaded by Emiliyan Petkov on 12 June 2015.

The user has requested enhancement of the downloaded file.

Емилиян Петков

ОСНОВИ НА ПРОГРАМИРАНЕТО СЪС С++

Велико Търново
2014

ОСНОВИ НА ПРОГРАМИРАНЕТО СЪС C++
Първо издание

Автор:
© Емилиян Георгиев Петков

Рецензент:
доц. д-р Георги Стоянов Тодоров

Коректор:
Радостина Каменова Петкова

© Издателство “Фабер”

ISBN:

Велико Търново, 2014

Съдържание

Предговор 6
Глава I. Въведение в програмирането на C++ 7
1.1. Програмиране 7
1.2. Машинен код 8
1.3. Езици за програмиране от високо ниво. 9
1.4. Алгоритъм. 9
1.5. Отговорности на програмиста. 12
1.6. Процесът на разработване на софтуер. 12
1.7. Създаване на проект с Code::Blocks 13
1.8. Компилиране на програма. 17
1.9. Структура на програма 17
Глава II. Типове данни 21
2.1. Данни. 21
2.2. Аритметични типове данни. 23
2.3. Тип псевдоним. 28
2.4. Тип изброим. 29
2.5. Синтаксис на езика. 30
Глава III. Входни и изходни потоци. Аритметични операции 31
3.1. Изходен поток cout. 32
3.2. Входен поток cin. 33
3.3. Аритметични операции. 34
Глава IV. Оператори за сравнения. Логически операции. Условен оператор „? :”. Побитови операции. Вградени функции 39
4.1. Оператори за сравнения. 39
4.2. Логически операции. 39
4.3. Условен оператор „?:”. 40
4.4. Побитови операции. 41
4.5. Оператори за присвояване на стойности. 44
4.6. Приоритети на операциите. 45
4.7. Вградени функции. 45
Глава V. Оператори за разклонения 50
5.1. Условен оператор if else. 50
5.2. Оператор switch. 56

Глава VI. Оператори while, do while и for 58
6.1. Цикъл while. 58
6.2. Цикъл do while. 60
6.3. Цикъл for. 62
6.4. Предусловие и постусловие. 65
6.5. Оператор goto. 66
6.6. Вложени цикли. 66
Глава VII. Указатели. Масиви 69
7.1. Указатели 69
7.2. Масиви. 73
Глава VIII. Динамична памет 79
8.1. Оператори new и delete. 79
8.2. Динамични масиви. 81
Глава IX. Масиви от символи 87
9.1. Дефиниране на масиви от символи. 87
9.2. Вградени функции в езика за работа с масиви от символи. 89
Глава X. Функции 95
10.1. Главна функция main. 95
10.2. Дефиниране на функции. 96
10.3. Деклариране на функции. 98
10.4. Предаване на параметри на функции по стойности. 99
10.5. Предаване на параметри на функции чрез псевдоними. 99
10.6. Предаване на параметри на функции по адреси. 100
10.7. Предаване на масиви като параметри на функции. 101
10.8. Област на действие на променливи и константи. 106
Глава XI. Указатели към функции. Разпределение на паметта.
Вградени и предефинирани функции. Рекурсия 108
11.1. Указатели към функции. 108
11.2. Разпределение на паметта. 109
11.3. Вградени функции. 112
11.4. Предефинирани функции. 112
11.5. Рекурсия. 112
Глава XII. Структури и класове 116
12.1. Структури. 116
12.2. Класове. 121
Глава XIII. Клас vector. Клас string 125
13.1. Клас vector. 125
13.2. Клас string. 129
Глава XIV. Файлове 134

14.1. Файлови потоци. 134
14.2. Текстови файлове. 136
14.3. Моментни местоположения при четене и запис. 138
14.4. Двоични файлове. 139
Глава XV. Модули. Тестване на програми 141
15.1. Модули. 141
15.2. Тестване на програми. 148
Литература 150

Предговор

Програмирането е както наука, така и професия, умение, което се усвоява чрез постепенно натрупване на знания и тяхното умело прилагане. Това означава, че в процеса на обучение, всяко нещо, което бъде научено, трябва да бъде приложено в програма, да се опита и да се види как работи на практика. Тези процеси на научаване и прилагане, а понякога и експериментиране, отнемат време.

Този курс е подготвен за студентите от специалностите „Компютърни науки" и „Информатика" във Факултет „Математика и информатика" на Великотърновския университет, изучаващи „Основи на програмирането". Лекциите в тази дисциплина са 30 часа на семестър, а упражненията – 60 часа (към настоящия момент). За да може материалът да бъде безпрепятствено усвоен от студентите са предвидени 150 часа извънаудиторна работа през семестъра, който се състои от 15 седмици. Това прави 10 часа на седмица самостоятелна работа на студент. Аз горещо препоръчвам на всеки студент, който иска да усвои основите на програмирането, да посещава всички лекции и упражнения, както и да отделя определеното допълнително време в разработване и писане на компютърни програми. Това ще гарантира неговия краен успех.

Бъдете последователни. Приемете тези петнадесет теми като петнадесет важни градивни елемента в основата на вашето изграждане като компютърни специалисти. Дори да не се чувствате призвани да се занимавате основно с програмиране, наученото тук ще ви бъде от голяма полза в усвояването на останалите предмети в специалността ви. А ако сте от тези, които желаят да създават софтуер, трябва да знаете, че колкото по-големи и прецизни направите основите на знанията и уменията си в програмирането, толкова по-голямо и високо „здание" ще можете да построите след това.

Глава I. Въведение в програмирането на С++

1.1. Програмиране.

В наши дни компютрите заемат централно място в изпълнението на различни процеси. На много места тяхното отсъствие е вече немислимо. Всеки компютър работи с *програми,* наричани още *софтуер* (от англ. software). Този учебник има за цел да въведе обучаващите се в процеса на разработка на компютърни програми, наречен *програмиране*. Разработването на софтуер в световен мащаб представлява най-големият отрасъл в компютърния бранш и заема основните позиции в компютърните науки.

Най-общо една компютърна система се състои от хардуер и софтуер. Ефективността ѝ зависи от успешната работа както на хардуера, така и на софтуера. Следователно, важно е още в самата основа на обучението да заложим правилните принципи в програмирането, които винаги ще бъдат главната предпоставка за създаването на висококачествен софтуер.

Разработката на софтуер основно се определя от задачите (проблемите), които се поставят за решаване. В този смисъл, програмирането е процесът на изразяване на решението на даден проблем по начин, по който компютърът може да го реши. Повечето от времето и труда в програмирането е именно намирането на правилно решение на поставената задача. Чрез програмирането ние прилагаме идеите си за решаване на дадена задача и използваме техники и образци, за да дадем на компютъра решението, което той да изпълни. Следователно, целта ни е да разгледаме методите и средствата, чрез които програмираме компютъра, за да изпълнява определени действия.

Има са много езици за програмиране, но един от най-универсалните и масово използваните е С++, създаден от Бярне Строуструп (Bjarne Stroustrup) [3, 5]. Към този момент езикът е използван като основа в разработването на много други по-специализирани езици за програмиране като Java, JavaScript, C# и много други. Този учебен курс има за цел да постави основите в програмирането като използва езика С++. Въпреки че принципите в

програмирането са универсални, решенията могат да бъдат разнообразни, а те много зависят от използвания език за програмиране. Така че тук освен общите принципи в програмирането, ще бъдат разгледани и основни идеи и техники в програмирането на C++.

1.2. Машинен код.

Процесорът на компютъра изпълнява *машинни инструкции*, наречени още *машинни команди* [7]. Например:

1. Премести съдържанието на клетка 2700 от ОП в регистъра AX.
2. Извади 2 от регистъра AX.
3. Ако резултатът е положителен, премини към инструкцията, която се намира в клетка 3000 от ОП.

Тези инструкции се представят като числа, за да могат да се съхраняват в паметта. Ето как горният пример ще се кодира за процесор Intel 80386:

```
161 2700 45 2 127 3000
```

За да се улесни писането на програми и за да се оптимизира процесът на записване на инструкциите в паметта, е създаден асемблерен език (на англ. assembly language). Всеки тип процесор си има свой асемблерен език. Този език е съвкупност от кратки имена, наречени мнемокодове, като всяко име отговаря на една машинна команда. Така писането на програми е по-разбираемо за човека, а и имената на командите се помнят по-лесно. Това значително улеснява писането. След като една програма е написана на асемблерен език, тя трябва да се преведе в поредица от машинни инструкции. Това се прави от програма, наречена Асемблер (Assembler). Асемблерният език е език от ниско ниво. Ето как би изглеждал горният пример, написан на асемблерен език:

```
MOV AX, [2700]
SUB AX, 2
JG 3000
```

Асемблерният език има два съществени недостатъка. Първият произлиза от факта, че типовете процесори са много и постоянно излизат нови с нови колекции от инструкции. Това означава, че програмите, написани за един процесор, няма да работят на процесор от друг тип. Вторият недостатък е, че при писането дори и на по-елементарни програми, инструкциите са много.

1.3. Езици за програмиране от високо ниво.

В търсеното на решения на тези два проблема се появяват езиците от *високо ниво*. Те започват да се разработват в началото на 50-те години [7]. В тези езици програмата се описва с оператори, които по-схематично описват задачата, която трябва да бъде решена. Описанието на програмата се нарича *програмен текст* и още *програмен код, първичен код* (от англ. source code). За превеждането на това описание в машинен код, се използва специална програма, наречена *компилатор* (от англ. compiler). Това прави езиците от високо ниво да са независими от хардуера. Ако има необходимост една програма да се изпълнява на различни типове процесори, тя трябва да се компилира с компилатори за тези процесори. Компилаторите превеждат една програма от език от високо ниво на машинен код, ако тя е написана според правилата на езика.

1.4. Алгоритъм.

Преди да реализираме решението на поставена задача чрез програма, трябва да обмислим и подредим елементарните действия, които компютърът ще изпълнява [1]. *Елементарно действие* е това, което може да се извърши без допълнителни пояснения, а *стъпка* се нарича изпълнението на елементарно действие. Последователността от стъпки, които описват програмата и водят до решение на задачата, се нарича *алгоритъм*. Понякога намирането на алгоритъм е лесно, но понякога това ще изисква задълбочен анализ, много познания (особено в математиката) и изобретателност.

1.4.1. Свойства на алгоритмите.

Всяка програма трябва да следва своя алгоритъм, защото само това ще доведе до решението на задачата. Писането на програмния текст се явява реализация на алгоритъма. За да може един алгоритъм да се реализира като програма, той трябва да притежава следните свойства:

- *Дискретност* – алгоритъмът се състои от краен брой елементарни действия.
- *Яснота* – стъпките са различими една от друга и еднозначно е определена последователността, в която те се изпълняват.
- *Крайност* – алгоритъмът завършва след изпълнението на краен брой стъпки.
- *Резултатност* – алгоритъмът винаги дава резултат, ако входните данни принадлежат на допустимото множество от такива данни.
- *Определеност* – при едни и същи входни данни се получава един и същи резултат.
- *Масовост* – алгоритъмът се прилага (е определен) за различни входни данни, принадлежащи на допустимото множество от такива данни (прилага се за решаване на коя да е задача от клас еднотипни задачи).
- *Формалност* – алгоритъмът може да се изпълнява формално докато достигне указание за край, т.е. не се изисква от изпълнителя да знае целта, която се преследва (може да се изпълнява от автомат).

Ефективността на алгоритмите е едно незадължително, допълнително свойство, което обаче често се изисква. Казваме, че един алгоритъм е ефективен, когато той се изпълнява за оптимален брой стъпки и използва възможно най-малко количество памет за временното съхраняване на данните, с които работи.

1.4.2. Видове алгоритми.

Според последователността на елементарните действия, които алгоритмите извършват, те могат да бъдат разделени на няколко вида. Ето основните от тях:

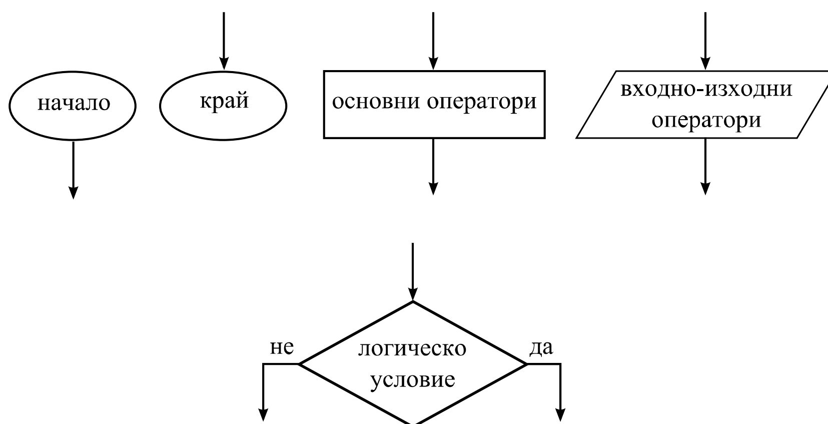
- *Линейни* – елементарните действия се изпълняват винаги последователно.
- *Разклонени* – конструкции, които в зависимост от поставени условия, определят различни пътища от елементарни действия, които да бъдат изпълнени.
- *Циклични* – конструкции, които в зависимост от поставени условия, изпълняват една стъпка или група от стъпки многократно.

- *Рекурсивни* – конструкции, в чийто стъпки се намират обръщения към самите тях.
- *Евристични* – използват специфична информация, свързана със задачата, която е резултат от практически и/или експертен опит. Характерно за тези алгоритми е, че те водят до решение, макар и то да не е оптималното. Реализацията им е по-проста в сравнение с алгоритъма на оптималното решение.
- *Вероятностни* – генерират решения, всяко от които се приема с някаква вероятност.
- *Паралелни* – реализират паралелно изпълнение на няколко различни процеса, като ги създават и след това обединяват получените от тях резултати.

1.4.3. Описание на алгоритми.

Описанието на алгоритмите се извършва основно чрез: текст (стъпките на алгоритъма се описват на формален език), графики (блок схеми), програмен текст и псевдокод (свободно съчинен език за описание на алгоритми, най-често се състои едновременно от текст и програмен текст).

Описанието на алгоритми чрез блок схеми се прави основно чрез графичните елементи, показани на фигура 1.1.



Фиг. 1.1. Основни графични елементи за описание на алгоритми.

1.5. Отговорности на програмиста.

Всяко цифрово устройство работи с програми. Ако само една програма или дори част от нея не работи според изискванията и очакванията на потребителите, тя ще създаде проблеми. Затова писането и внедряването на програми е отговорна задача [1, 2]. Едно от най-важните качества, които трябва да притежават програмите ни, е *точност*. Това означава, че програмата извършва точно това, което искаме и очакваме от нея – тя не решава само част от поставените й задачи и не изпълнява странични (нежелани) задачи. Така програмата става *надеждна*. Програмите ни не трябва да създават неприятности на потребителите, а да бъдат полезни и надеждни. Софтуерът също трябва да бъде *добре проектиран*. Той трябва да предоставя добри условия за работа с него, а не да дразни потребителите с неговата недобра последователност от задачи, мудност и др. Добре е също кодът на програмите, които пишем, да бъде *добре структуриран*. Това е предпоставка, той да бъде лесно разбираем както от нас, така и от наши колеги. Това например ще позволи и други да участват в развитието на нашата програма.

1.6. Процесът на разработване на софтуер.

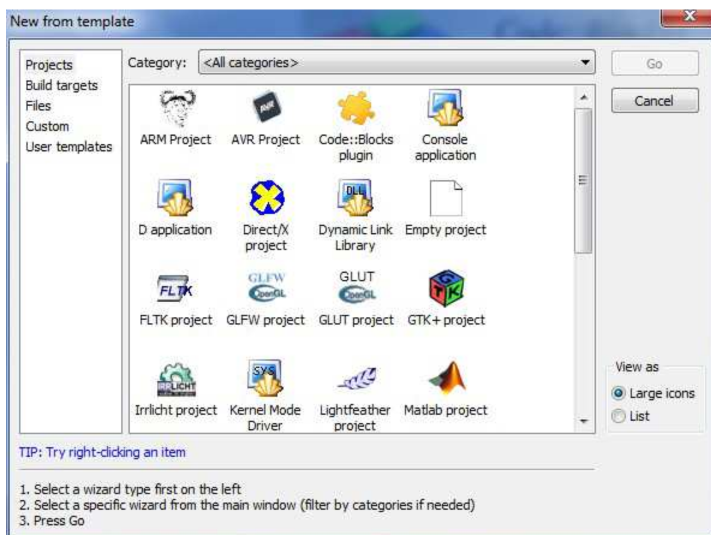
Процесът на разработване на софтуер обикновено се описва със следните четири етапа [1]:

- *Анализ* – описанието на задачата; разбирането от какво се нуждае потребителят и какво иска да получи.
- *Проектиране* – определяне структурата на програмата; частите, от които програмата ще се състои; начинът на свързване на частите една с друга и начинът, по който програмата комуникира с потребителя.
- *Кодиране* (*програмиране*) – писане на програмния текст (изразяване на решението в код) според всички поставени изисквания.
- *Тестване* – прилагане на предварително подбрани тестове, които доказват точността и надеждността на програмата.

Много често последните два етапа се наричат *реализация* (от англ. implementation).

1.7. Създаване на проект с Code::Blocks.

Приложението Code::Blocks е интерактивна среда за разработка на програми (от англ. Interactive Development Environment – IDE), които се пишат на C++. Компилаторът, който се използва по подразбиране, е включеният в GNU[1] Compiler Collection (GCC) за езика C++. Основните елементи от графичния интерфейс на Code::Blocks са менюта, ленти с бутони, панел за управление, панел за редактиране на програмните текстове и панел за доклада след компилирането. Програмните текстове в това учебно пособие са писани и компилирани с Code::Blocks версия 10.05. Те са от конзолен тип т.е. извеждането и въвеждането на данни се прави чрез командния интерпретатор (конзолата) на съответната операционна система.



Фиг. 1.2. – Избор на тип на нов проект.

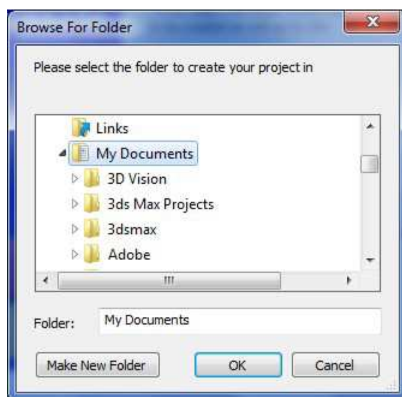
[1] GNU е свободен софтуер, проект на широко сътрудничество, обявен на 27 септември 1983 г. от Ричард Столман в Масачузетския технологичен институт. Целта е да се даде свобода и контрол на компютърните потребители в използването на техните компютри и компютърни устройства чрез съвместно разработване и осигуряване на софтуер, който се основава на следните права за свобода: потребителите са свободни да стартират софтуера, да го споделят (копират, разпространяват), изучават и модифицират.

Всички програми, които пишем в тази среда, се реализират като проекти. Проектът съдържа програмните текстове, записани във файлове, проектния файл и евентуално хедър файлове (от англ. header files), мейк файл (от англ. make file), изпълним файл и др. Създаването на проект може да стане от менюто File като се избере New -> Project. Ще се появи диалоговият прозорец, показан на фигура 1.2. Трябва да маркираме Console application и да натиснем бутона Go. На следващата стъпка трябва да изберем езика C++ (фиг. 1.3).

Фиг. 1.3. – Избор на език (C или C++).

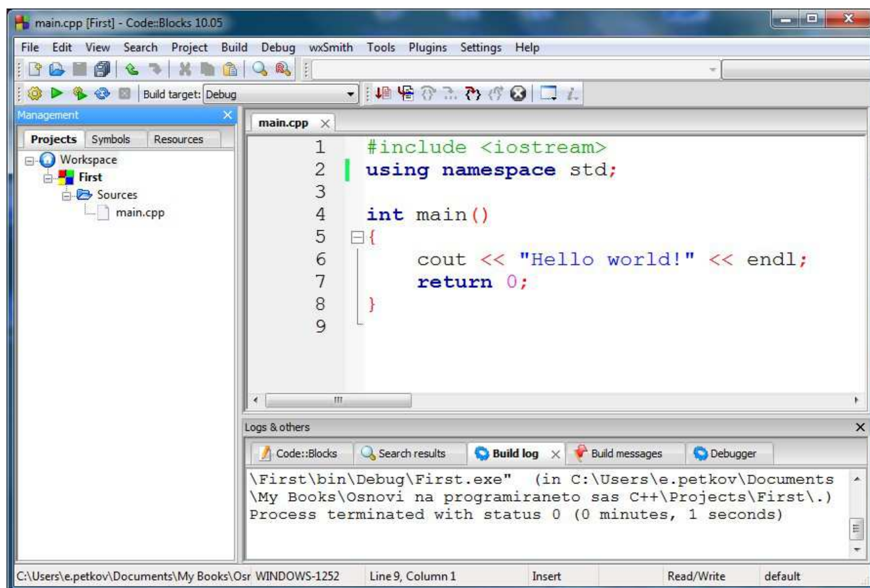
Фиг. 1.4. – Избор на име на проекта.

На следващия диалогов прозорец трябва да посочим име на проекта (фиг. 1.4) и директория, в която той да бъде записан (фиг. 1.5).



Фиг. 1.5. – Избор на директория на проекта.

Фиг. 1.6. – Избор на компилатор и директории на изпълнимите файлове.



Фиг. 1.7. – Проектът е създаден. Кратък програмен текст е генериран в main.cpp.

На последната стъпка избираме компилатор (GNU GCC Compiler) и имената на папките за изпълнимите файлове, които той ще създава (фиг. 1.6) в двата вида конфигурации при компилиране (достатъчно е да се отбележи само една конфигурация, например Debug). След изпълнението на тези стъпки, проектът е готов. Кратък програмен текст е генериран в main.cpp (фиг. 1.7).

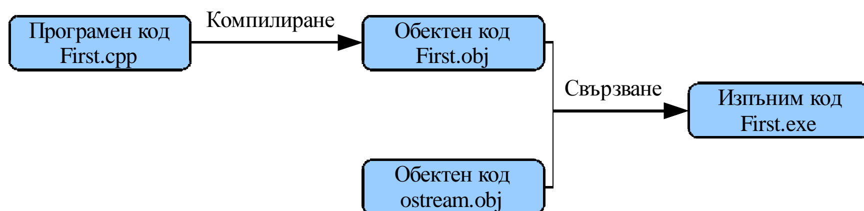
Нека да разгледаме папките и файловете в тях, които Code::Blocks създава като част от проекта. Първо се създава една главна папка (в посочената от нас директория), която носи името на проекта First и в която се записват: First.cbp – проектния файл (Code::Blocks project), main.cpp – програмния текст, First.depend – файл, описващ зависимости и First.layout – файл, описващ оформление. В случай на избрана Debug конфигурация се създават папките: obj\Debug, където се записва main.o и bin\Debug, където се записва First.exe (за Windows). Ако е избрана Release конфигурация се създават папките: obj\Release, където се записва main.o и bin\Release, където се записва First.exe (за Windows).

1.8. Компилиране на програма.

За да може програмата, която е написана на C++, да се изпълни от компютъра, тя трябва първо да бъде проверена за грешки и ако няма такива, програмният ѝ текст да бъде преобразуван (преведен) до машинен. Този процес се нарича *компилиране* (от англ. compiling). Програмният текст се записва в текстов файл обикновено с разширение (още нар. суфикс, от англ. suffix) **.cpp** (съкр. от C pluss pluss). След успешно завършен процес на компилиране се получава *обектен файл*. Обектният файл съдържа *обектен код*, който представлява машинен код, неготов за директно изпълнение, метаданни, описания на връзки между модули, коментари, програмни символи, информация за тестващата програма и др. Тази информация е тясно свързана с конкретната операционна система, под която се компилира. Обектните файлове обикновено носят името на файла с програмния текст и са с разширения .obj или .o.

Програмирайки на C++ ние можем да използваме вече разработени и компилирани модули. Стандартно някои от тях се предоставят като библиотеки, а други като обектни файлове. Например, описанието на изходния поток `cout` от `iostream` се намира в `ostream.obj`. Така за една програма, разположена в един файл (като от примера по-горе), могат да бъдат необходими няколко

обектни файла. Те заедно ще образуват изпълнимата програма, която представлява един изпълним файл под дадената операционна система. Създаването на този файл в C++ се прави от специална програма наречена *линкер* (от англ. linker), а процеса наричаме *свързване* (от англ. linking). В Wndows тези файлове се наричат *приложения* (от англ. applications) и са с разширения **.exe** (съкр. от executable).



Фиг. 1.8. – Примерна схема за изграждане на изпълним файл в C++.

В съвременните среди за разработка на приложения компилирането и свързването се обединяват в един процес наречен *изграждане* (от англ. building), а командата обикновено е Build. На фигура 1.8 е дадена примерна схема на изграждане на изпълним файл.

Процесът на компилиране вместо с обектен файл може да завърши с доклад за грешки. Тези грешки се наричат *грешки по време на компилиране* (от англ. compile-time errors). Това най-често са синтактични грешки. Компилаторът може още да генерира и *предупреждения* (от англ. warnings). Те най-вече са свързани с неявното преобразуване на типове. Предупрежденията не възпрепятстват изграждането на изпълнимия файл, но трябва да бъдат взети под внимание от програмиста.

Грешки могат да се появят и в свързването. Те се наричат *грешки по време на свързване* (от. англ. link-time errors). С грешки по време на компилиране и грешки по време на свързване изпълним код не е възможно да бъде изграден.

Възможна е появата на грешки и по време на изпълнение на програмата. Те се наричат *логически грешки* (от англ. logic errors) или още *грешки по време на изпълнение* (от англ. run-time errors). Тези грешки могат да бъдат открити само при изпълнението на програмата. Този процес е част от *тестването* на програмата. Това

обикновено са логически неточности, неспазване на поставени изисквания и др. Затова те още се наричат *семантични грешки*. Семантиката разглежда значението на програмата.

Всяка съвременна среда за разработка на програми разполага с *дебъгер* (от англ. debugger). Това е програма, с чиято помощ програмистът по-лесно открива и коригира грешки.

Следователно, основните елементи в интерактивните среди за разработка на програми са текстов редактор, компилатор, линкер и дебъгер.

1.9. Структура на програма.

Първата програма, която ще разгледаме, е тази, която средата за разработка генерира. Нека първо да разгледаме кода, а след това всеки ред ще бъде обяснен:

```
//програма, извеждаща Hello world! в командния
//интерпретатор.
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Двете наклонени черти надясно (//) указват на компилатора, че следват символи на реда, които той трябва да игнорира. Операторът // действа до края на реда. Двете черти заедно с текста след тях се наричат *коментар*. Чрез него в тази програма се дава кратко пояснение за нейната цел. Ако коментарът трябва да е по-кратък от един ред или да се простира на няколко реда, тогава може да се използва и конструкцията `/* коментар */`. Тези оператори указват на компилатора, че трябва да игнорира текста.

Всяка ключова дума, която започва със знака #, се нарича директива на препроцесора. *Препроцесорът* предоставя възможността за вмъкването на хедър файлове, макроси и др. в програмата. Препроцесорът се изпълнява преди да започне компилирането на кода. Директивата `#include` указва на компилатора да направи достъпни (да вмъкне) различни помощни

средства. В случая iostream (съкр. от **i**nput **o**utput **stream** – входно-изходен поток) прави възможна употребата на входно-изходните потоци такива като cout и cin.

Думата using указва на компилатора, че той трябва да въведе имената от дадено наименувано пространство (namespace), което представлява контейнер за множество от запазени имена в C++. Наименуваното пространство std включва имената на всички дефинирани компоненти в езика. Този код е необходим заради iostream.

Всяка програма на езика C++ трябва да съдържа една основна структура, наречена *главна функция* main. Това име е задължително. В програмите на C++ можем да имаме много такива структури, наречени *функции*, с разнообразни имена, но изпълнението на програмата започва от main. Както всяка функция, така и тази трябва да има тип на върнатата стойност (в сл. int), име main, две скоби (), в които може да има параметри и тяло. Тялото на нашата функция е:

```
{
    cout << "Hello world!" << endl;
    return 0;
}
```

То се състои от действия (оператори), поставени във фигурни скоби {}. В нашия случай имаме два оператора, които се изпълняват: cout и return. Операторът cout е съкращение от **c**haracter **out**put stream (изходен поток от символи) и служи за извеждането на символни данни обикновено в командния интерпретатор (command prompt при Windows и shell при Unix и Linux). "Hello world!" представлява символен низ, endl – константа, съдържаща символа за край на ред '\n' и return – прекъсва изпълнението на функцията, връща управлението на операционната система, като й подава числова стойност 0, означаваща, че програмата е завършила успешно.

Глава II. Типове данни

Типовете данни в програмирането представляват спецификация на информацията, която трябва да се обработи от компютъра [6].

Типовете данни в програмирането представляват механизъм за обобщено описание на групи от данни (обекти), с които даден език за програмиране борави с цел те да станат „познати" и различими за компилатора. Всеки тип данни определя следните атрибути на обектите:

- диапазон от допустими стойности;
- обем от оперативната памет, който те заемат;
- допустим набор от операции, които могат да се прилагат върху тях.

Последното в повечето случаи се подразбира, без да се посочва в явен вид. Например за целите числа се прилагат всички аритметични операции, включително целочислено деление и деление по зададен модул.

Концепцията на типовете данни има важна роля за работата на компилатора и изпълнението на програмите от гледна точка на контрол на допустимост на стойностите на данните, оптимизация на заеманата памет, надеждност на програмите, дефиниране на потребителски (нестандартни или още абстрактни) типове данни и др.

2.1. Данни.

Компютърната система работи с нули и единици. Най-малката област в паметта, която може да бъде четена и записвана наведнъж е един байт (8 бита). Могат да бъдат записвани поредица от нули и единици, организирани в байтове. Тези поредици от нули и единици могат да бъдат интерпретирани различно. Например, числото $10000001_{(2)}$, записано в паметта, в някои случаи се интерпретира като $-1_{(10)}$, а в други като $129_{(10)}$. За всяка стойност, която се записва в паметта на компютъра, трябва да бъде определен

начин на интерпретиране т.е. какви данни тази стойност представлява.

В паметта на компютърните системи можем да записваме само числа (в двоична бройна система). Първото разделение, което е направено на числовите данни, е на цели и дробни (реални) числа. Компютърът може да работи и с двата типа числа. Но също така знаем, че той съхранява и обработва текст или поредици от символи. Начинът за представяне на символи в компютъра е чрез числово кодиране на всички символи, които се използват. Например, числото $11000001_{(2)}$, записано в паметта, може да се интерпретира като $97_{(10)}$, а и като символът 'a' от латинската азбука.

Програмите обработват данни: числови стойности, указващи количества, суми, тежести, относителни дялове, номера; текстови – наименования, съобщения, текстове; логически; графични; съставни и др. За да могат тези различни видове данни да се съхраняват в паметта на компютъра и да се обработват от програми, в C++ съществуват различни *типове данни*.

Съхраняването на данните, с които една програма работи, става в поле от паметта с определен брой байтове, наречено *обект* (от англ. object) [4]. Следователно, обектът е регион от паметта, свързан с тип на данните и определящ какви данни могат да бъдат записвани там. В програмите всеки обект получава *име* наричано още *идентификатор* (от англ. identifier). Обектите могат да бъдат *модифицируеми* и *немодифицируеми*. Модифицируемите се наричат *променливи* (от англ. variables). На тези обекти можем да задаваме различни стойности по всяко време на жизнения им период. Немодифицируемите обекти се наричат *константи* (от англ. constants). Техните стойности, веднъж зададени, не могат вече да бъдат променяни. За да се използват променливи и константи в програма първо трябва да се *дефинират*.

При дефиниране на променлива се указва нейният тип и нейният идентификатор. Това в C++ може да се зададе със следния синтаксис:

```
<name_of_type> <identifier> [ = <expression>];
```

При дефиниране на константа се използва ключовата дума `const`, указва се нейният тип, нейният идентификатор и стойността,

която тя приема. Това в C++ може да се зададе със следния синтаксис:

```
const <name_of_type> <identifier> = <expression>;
```

Ако типът на константа не бъде указан се подразбира един от типовете в езика – `int`. Дефинирането на константи в C++ може да стане и чрез директивата на препроцесора `#define`. Тя служи също и за дефинирането на макроси в програмите. Ето един пример за дефиниране на константи:

```
#define Pi  3.14159
#define Sum 0.0
```

В програмите могат да бъдат *декларирани* променливи, константи, функции и др., т.е. указва се, че те са дефинирани вече някъде в програмата (евентуално във външен файл) или ще бъдат дефинирани по-късно. Декларациите не създават обекти в паметта.

Идентификаторите в C++ могат да се състоят от букви от латинската азбука (малки и големи), цифри и подчертаващо тире (_), като не могат да започват с цифра. В идентификаторите в C++ малките и големите букви са различими една от друга, например A е различно от a. Ето няколко примера на идентификатори:

```
Pi, abs, S1, S2, number_of_characters.
```

В тази глава ще се спрем на някои *скаларни типове данни*. Това ще бъдат *аритметичните типове данни*, *тип на данни псевдоним* и *тип на данни изброим*.

2.2. Аритметични типове данни.

Първото предназначение на компютрите е било да работят като изчислителни машини. Най-важните типове данни в програмите са аритметичните. Аритметичните типове представят цели и реални числа. Чрез тези типове данни програмите могат да съхраняват и обработват цели и реални числа. Например, числата 0, 5, -3 са цели, а числата 3.14159, 0.2, -5.0 са реални. Начинът на представяне на целите и реалните числа в компютъра е различен.

В таблица 2.1 и таблица 2.2 са дадени аритметичните типове в C++. В таблица 2.1 са дадени типовете за представяне на целите числа, а в таблица 2.2 – реалните. Трябва да се отбележи, че съществуват компилатори, при които броят битове за представяне на числата за някои типове е различен от дадените стойности в таблиците. Освен това, диапазоните за представяне на реалните числа зависят и от конкретната система, на която се компилира програмата.

Типът на данни `bool` се нарича логически. Той служи за представяне на логически стойности и изрази. Използват се стойностите 0 и 1 за кодиране на логическите стойности съответно „лъжа" и „истина". Например, резултатът от сравненията трябва да бъде „лъжа" или „истина". Ако напишем израза 5 > 7, то компютърът трябва да ни отговори с „лъжа" или това е числовата стойност 0. Ако напишем израза 5 < 7, то компютърът трябва да ни отговори с „истина" или това е числовата стойност 1. Логическият тип на данни bool е наречен така на името на изобретателя на математиката, базирана на 0 или 1 (булева алгебра), ирландския математик Джордж Бул (George Boole) (1815-1864).

Таблица 2.1. Типиве данни за представяне на цели числа.

Тип	**Количество битове (байтове)**	**Диапазон за представяне на числата**
`bool`	8 (1)	0 и 1
`char`	8 (1)	$от\ \ -2^7\ \ до\ \ 2^7-1$, т.е. [-128 $\div$ 127]
`short`	16 (2)	$от\ \ -2^{15}\ \ до\ \ 2^{15}-1$, т.е. [-32768 $\div$ 32767]
`int`	16 (2) или 32 (4)	$от\ \ -2^{15}\ \ до\ \ 2^{15}-1$, т.е. [-32768 $\div$ 32767] или $от\ \ -2^{31}\ \ до\ \ 2^{31}-1$, т.е. [-2147483648 $\div$ 2147483647]
`long`	32 (4)	$от\ \ -2^{31}\ \ до\ \ 2^{31}-1$, т.е. [-2147483648 $\div$ 2147483647]
`unsigned char`	8 (1)	$от\ \ 0\ \ до\ \ 2^8-1$,

		т.е. [0 ÷ 255]
`unsigned short`	16 (2)	*от* 0 *до* $2^{16}-1$, т.е. [0 ÷ 65535]
`unsigned int`	16 (2), 32 (4)	*от* 0 *до* $2^{16}-1$, т.е. [0 ÷ 65535] или *от* 0 *до* $2^{32}-1$, т.е. [0 ÷ 4294967295]
`unsigned long`	32 (4)	*от* 0 *до* $2^{32}-1$, т.е. [0 ÷ 4294967295]
`unsigned long long`	64 (8)	*от* 0 *до* $2^{64}-1$, т.е. [0 ÷ 18446744073709551615]

За конкретна компютърна система и компилатор можем да разберем количеството байтове, необходими за представяне на числата, чрез оператора `sizeof(` *`type`* `or` *`variable`*`)` за всеки стандартен тип в C++. Например, можем да напишем:

```
cout << sizeof( int ) << endl;
или
unsigned long long l = 0;
cout << sizeof( l ) << endl;
```

След като вече са дадени някои от типовете данни в езика, можем да разгледаме пример за дефиниране на променливи и константи от различни типове. Следващите редове демонстрират дефинирането на няколко променливи и константи:

```
const double pi = 3.14159;
const int i( 0 );
char c;
int a, b, c, d = 0;
```

При дефиниране на променливи повечето компилатори не им задават начални стойности, например 0. Това обикновено е грижа на програмиста. Този, който разработва програмата, би трябвало да направи така, че всички променливи да приемат правилните стойности. Добър стил на програмиране е да се задават подходящи

стойности на променливите още при тяхното дефиниране. Това спомага за допускането на по-малко семантични грешки. Например, ако са необходими три променливи за въвеждане на три числови стойности в програма, чрез които да се изчисли нова стойност по някаква формула, променливите би трябвало да се дефинират така:

```
double a = 0.0, b = 0.0, c = 0.0, value = 0.0;
```

Операраторът = означава *присвоява стойност*. При дефиниране на променлива или константа нейната стойност може да бъде инициализирана и по този начин:

```
<name_of_type> <identifier> [ ( <expression> ) ];
```

Нека обясним значението на израза с пример:

```
short step = 3; или short step( 3 );
```

Той означава, че: 1) променливата ще бъде свързана с множеството от допустими стойности на типа `short` 2) създава се обект в паметта с количеството, в което може да се запише най-голямата стойност от този тип (2 байта) 3) на променливата се дава име `step` 4) присвоява се стойност цялото число $3_{(10)}$, което означава, че в паметта се записва числото $00000000\ 00000011_{(2)}$.

Ако променливи и константи са дефинирани извън функциите в програмата, те се наричат *глобални*. Обикновено глобалните променливи и константи се дефинират в началото на реализацията (програмния текст). Така те са видими за всички функции, дефинирани по-долу в програмата.

В таблица 2.2 са дадени типовете данни за представяне на реалните числа в езика C++.

Таблица 2.2. Типове данни за представяне на реални числа.

Тип	**Количество битове (байтове)**	**Диапазон за представяне на числата**
`float`	32 (4)	с точност до 7-ия знак от -3.4e+38 до -3.4e-38; 0.0; и от

		3.4e-38 до 3.4e+38
`double`	64 (8)	с точност до 14-ия знак от -1.7e+308 до -1.7e-308; 0.0; и от 1.7e-308 до 1.7e+308
`long double`	96 (12)	с точност до 21-вия знак

Точността при представянето на реалните числа означава брой символи за записване на дробната част. Например в тип `float` обикновено могат да се записват до 7 символа, т.е. точката плюс още шест. В случай че подаденото число е с повече цифри, тогава повечето компилатори извършват закръгляне до шестата цифра след точката. Например, ако в една програма запишем

```
float pi_f =  3.14159265359;
```

то повечето компилатори в pi_f ще запишат числото 3.141593. Ето един пример:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float pi_f = 3.14159265359;
    cout << setprecision(7)
            << "pi_f="<< pi_f << endl;
    cout << “float is “;
    cout << sizeof( float )
            << “bytes.\n”;
    int i;
    cout << “int is “ << sizeof( i )
            << “ bytes.\n”;
    cout << "END OF PROGRAM!\a" << endl;
    return 0;
}
```

Текстовете, поставени между две кавички се наричат символни низове. Знакът `‘\n’`, поставен в символния низ, е със специално предназначение. Той указва на потока да вмъкне нов ред

на мястото, където той се среща. Този знак е присвоен на константата `endl` (съкр. от end of line). В C++ можем да ползваме и други специални знаци, които се поставят в символни низове. Те се наричат още символни константи и са дадени в таблица 2.3.

Таблица 2.3. Специални знаци.

Знак	**Действие в потока**
\n	Вмъква нов ред
\r	Връща маркера в началото на текущия ред
\t	Вмъква табулация
\v	Вмъква вертикална табулация
\b	Изтрива символа вляво
\f	Вмъква нова страница (на принтера)
\a	Вмъква звуков сигнал
\'	Вмъква апостроф
\"	Вмъква кавички
\?	Вмъква въпросителен знак
\\	Вмъква наклонена черта наляво

2.3. Тип псевдоним.

В езика C++ е въведен типа *псевдоним* (от англ. reference). Чрез този тип става дефинирането на допълнителни имена (псевдоними) на вече съществуващи обекти в програмата. Псевдонимите се използват много ефективно при предаването на параметри на функции. При дефинирането на псевдоним се използва знакът `&`. Ето един пример:

```
double  ratio_of_l_to_d_of_a_circle = 3.14159265;
double& pi = ratio_of_l_to_d_of_a_circle;
```

Променливата `pi` става псевдоним на променливата `ratio_of_l_to_d_of_a_circle`. Това означава, че и двете имена се отнасят за един и същи обект в паметта. Променливата `ratio_of_l_to_d_of_a_circle` се нарича *инициализатор*. Псевдонимите не могат да бъдат създавани без наличието на инициализатор.

2.4. Тип изброим.

Изброим тип на данни се дефинира в програма чрез ключовата дума `enum`. Типът представлява множество от идентификатори, съответстващи на целочислени константи от типа `int`. Синтаксисът е следният:

```
<enum> <name_of_the_new_type>
      {     <identifier_1> [ = <expression>],
            <identifier_2> [ = <expression>],
             . . .
            <identifier_n> [ = <expression>]
      }
```

Чрез следващия програмен текст се дефинира изброим тип, представляващ дните от седмицата:

```
enum daysofweek
      { mon, tue, wed, thu, fri, sut, sun };
```

Всеки идентификатор от множеството на типа `daysofweek` представлява константа, на която е присвоена целочислена стойност. По подразбиране стойността на първата константа от списъка е 0, а всяка следваща приема инкрементираната стойност на предходната. Тези стойности също могат да бъдат определени от програмиста при дефинирането на типа. Например:

```
enum daysofweek
      { mon = 1, tue, wed, thu, fri, sut, sun };
```

Променливите, дефинирани от типа, могат да присвояват стойностите на всички константи от този тип, но не и от друг изброим тип. Изброим тип може да бъде дефиниран и като анонимен. В този случай променливите от този тип трябва да бъдат дефинирани веднага след дефиницията на типа. Ето един пример, който демонстрира тези особености:

```
daysofweek day;
day = mon;
enum
{ black, red=7, green, blue, white=15 } color;
```

```
color = white;
day = black; //error!
color = day; //error!
```

2.5. Синтаксис на езика.

Както всички формални езици, така и езиците за програмиране имат своя *азбука* – множеството от допустимите символи. Чрез азбуката на езика C++ и по определени правила са съставени думите на езика (лексика). А за да могат да се пишат изречения на този език, са съставени граматически правила.

Азбуката на езика C++ се състои от малките и големите латински букви, цифрите и специални символи, като някои от тях са: +, -, *, §, %, $, ?, [, { и др. За разделители в езика се използват символите за интервал, табулация и нов ред.

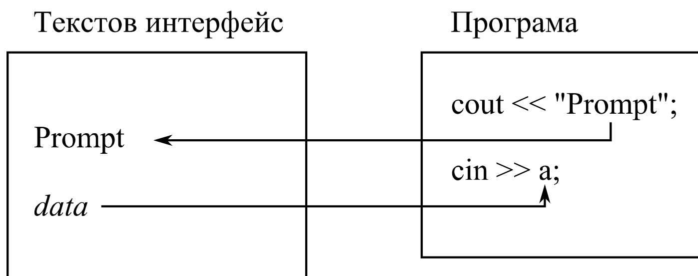
Думите в C++ могат да бъдат разделени на няколко вида: *ключови думи*, *оператори*, *препинателни знаци*, *идентификатори*, *числови и символни константи* и *символни низове*. Символни низове, числови и символни константи, идентификатори и препинателни знаци вече бяха използвани в примерите по-горе и бяха обяснени.

Операторите служат за пресмятания и за управление. Тези за пресмятания се наричат *аритметични оператори*, а тези за управление – *управляващи оператори*. Някои оператори се състоят от един символ, а други се обозначават с дума. Примери за аритметични оператори са: `+`, `++`, `-`, `--`, `/`, `%`, `*`, `&`, `|`, `and`, `or` и др. Примери за управляващи оператори са: `if…else`, `switch`, `for`, `while`, `do…while`, `break`, `new`, `delete` и др.

Глава III. Входни и изходни потоци. Аритметични операции

Основното предназначение на компютърните програми е да получават данни, да ги обработват и да извеждат резултати, състоящи се отново от данни [4]. Тази задача изисква от програмния език да разполага със средства за вход и изход на такава класифицирана информация. Това в езика C++ е организирано чрез *потоци* (от англ. streams). Потоците представляват екземпляри на класове, които от своя страна са свързани в сложни взаимовръзки в библиотеката `iostream`. Потоците служат за входни и изходни операции, файлови операции (в този случай се дефинират като екземпляри на определени за целта класове от йерархията на класа `iostream`), прехвърляне на данни между различни области на паметта и др.

В тази глава ще се спрем само на операторите `cout` и `cin`. Те са дефинирани съответно за изход и вход в и от команден интерпретатор, текстов терминал, конзолата на Windows, терминала на Mac OS X или по-общо казано *текстов интерфейс*. Въвеждането на данни със `cin` в програмата се осъществява само чрез клавиатурата. На фигура 3.1 е дадена примерна схема на действието на операторите `cout <<` и `cin >>`.



Фиг. 3.1. Въвеждане и извеждане на данни със `cin` и `cout`.

3.1. Изходен поток `cout`.

Потокът `cout` извежда в текстовия интерфейс на дадената операционна система данни от стандартните типове в езика. Това са цели числа, реални числа, символни низове и др. Дефиниран е да разпознава типовете самостоятелно. При извеждане на цели и реални числа, те се преобразуват до десетична бройна система (*dec*), ако не е зададена друга (*hex* – шестнадесетична, *oct* – осмична). Символните низове се извеждат без някакво преобразуване.

Възможно е известно управление (от англ. manipulation) на изходния поток. Това се прави с манипулаторите на входа и изхода (от англ. input-output manipulators): `setw`, `fixed`, `setprecision`, `setbase` и др. Достъпни са чрез наименуваното пространство `iomanip`. Манипулаторът `setw(` *pos* `)` задава *pos* на брой позиции в текстовия интерфейс, където да бъдат изведени данните като ги подравнява вдясно. Манипулаторът `setprecision(` *n* `)` задава десетична точност при форматиране на числа с плаваща запетая при извеждане. Параметърът *n* означава брой на цифрите в числото или при фиксиране на знаците (с манипулатора `fixed`) след десетичната точка – брой на тези цифри. Манипулаторът `setbase(` *base* `)` задава основата, в която числата да се извеждат. Ето един пример:

```
cout << setprecision( 5 ) << 125.2 << " $\n";
cout << fixed;
cout << setw( 15 ) << setprecision( 2 )
      << 525.25 << " $\n";
cout << setw( 15 ) << setprecision( 2 )
      << 120034.5 << " $\n";
cout << setbase( 16 ) << 166 << endl;

Изход:
125.2 $
         525.25 $
      120034.50 $
a6
```

При извеждане на данни много често се налага вмъкването в потока на знака за край на ред (`'\n'` или константата `endl`).

3.2. Входен поток `cin`.

Потокът `cin` служи за въвеждане на данни в програмата. Данните се въвеждат от клавиатурата и представляват поредици от символи. Тези символи трябва да бъдат интерпретирани по правилен начин. Например, ако трябва да се въведе реално десетично число чрез потока за вход, то трябва да се напише в текстовия интерфейс като поредица от символи за цифри и точка (пример: 3.14159265), след това тази поредици от символи да се преобразува до десетичното реално число. В друг случай може да се налага въвеждането на символен низ. Тогава въведените символи просто се прехвърлят по начина, по който са въведени.

Данните, които се въвеждат чрез `cin`, трябва да бъдат записани в обект от паметта, за да бъдат съхранени и използвани в програмата. Потокът `cin` изисква идентификатор на обект, т.е. променлива. Входният поток от символи се интерпретира в зависимост от типа на променливата, в която се записва. Например, ако променливата е от тип `int`, постъпващите в потока символи се преобразуват до цяло десетично число. Ако преобразуването е успешно, данните се записват в обекта на променливата.

Операторът `cin >>` предизвиква спиране на програмата, предоставяне на потребителя да въведе данни, след което при натискане на <enter> от страна на потребителя, данните се преобразуват до съответния тип и се записват в дадената променлива. С това работата на оператора завършва и програмата продължава своята работа.

Съществуват случаи, в които потокът от символи не може да бъде преобразуван до типа на променливата, в която трябва да се запише. Например, когато променливата е от тип `int`, а постъпващите символи са латински букви. В този случай данните остават невъведени, а променливата със старата си стойност. За щастие C++ разполага с механизми за откриване на тези грешки по време на изпълнение на програмата. В тези случаи казваме, че програмата получава некоректни данни от потребителя. Ето един пример на програмен текст, с който можем да откриваме некоректен потребителски вход:

```
int n = 0;
cin >> n;
```

```
if( not cin ) {
    cout   <<   "Incorrect   input!   The   program
              expected integer! n = 0!\n";
    cin.clear();
}
```

Добър стил на програмиране е, ако преди извикването на `cin` се използва изходният поток `cout`, за да се укаже на потребителя какви данни трябва да въведе. Това се прави с текстово съобщение, наречено *подсказка* (от англ. prompt).

Чрез едно извикване на `cin` е възможно въвеждането на множество от данни. Тогава те трябва да бъдат разделени с интервали. Потокът разпознава интервалите, табулациите и знаците за край на ред, въведени в текстовия интерфейс като разделители.

Ето един пример за извеждане на подсказка и въвеждане на стойностите на три променливи:

```
int a, b, c;
cout << "Please, enter values for a, b and c: ";
cin >> a >> b >> c;

Вход:
5  34  0  <enter>
```

3.3. Аритметични операции.

Аритметичните операции в езика C++, които се извършват между два операнда (бинарни операции), са: събиране (+), изваждане (-), умножение (*), деление (/) и целочислено деление с остатък (%). Операциите, които се извършват върху един операнд (унарни), са: унарен плюс (+), унарен минус (-), увеличаване на стойността с единица (инкрементиране) (++) и намаляване на стойността с единица (декрементиране) (--).

Аритметичните операции са дефинирани да работят с променливи, константи и изрази от стандартните аритметични типове данни в езика. Съществуват обаче някои особености, които тук ще бъдат разгледани.

В един аритметичен израз могат да участват променливи и константи от един и същи тип на данни. Тогава резултатът е от същия този тип. Например, типът на данни на целия израз, който е

поставен в потока cout в следващия пример, ще бъде от тип int, защото всички операнди са от тип int:

```
int a = 5, b = 10;
cout << (a + b) * 10 + a * 20;
```

В C++ в един аритметичен израз могат да участват променливи и константи от различните аритметични типове данни. Тогава типът на данни на целия израз се преобразува до „най-големия” тип. В таблица 3.1 са дадени правилата, по които се преобразуват типовете.

Таблица 3.1. Правила, по които се преобразуват типовете в C++.

Тип на данни	**Преобразува се до тип на данни**
char	int
unsigned char	int
short	int
unsigned short	unsigned int
enum	int
float	double

Също така трябва да се има предвид, че ако в една операция участват два операнда, като единият е от целочислен тип, а другият е от реален, тогава целочислената стойност се преобразува до реално число в типа на реалния операнд. В следващия пример променливата i е целочислена, f – реална от типа float и d – реална от типа double. За да се извърши операцията събиране, стойността на i ще се преобразува до типа float, след това ще се извърши събирането, след което, за да се извърши присвояването, резултатът от израза (число във float) ще се преобразува до типа double и тогава ще се присвои на d.

```
int i = 1; float f = 5.56; double d = 0.0;
d = f + i;
```

В езикът C++ е позволено и преобразуването на типовете от „по-големите” типове към „по-малките”, но в този случай е възможна загуба на информация. Например, ако искаме да присвоим стойност от тип double в променлива от тип int, ще загубим дробната част,

ако има такава и ако цялата част на числото надхвърля допустимите стойности за типа `int`, то и тя няма да може да се запише. В такива случаи компилаторът извежда предупреждения (warnings). В следващия програмен текст са дадени трите варианта, които могат да се случат, при такова преобразуване. В първия вариант имаме загуба на дробната част на реално число, във втория – нямаме загуба, в третия – губим и дробната част и цялата част на числото, като в променливата `sh` се записва максималната допустима стойност за типа `unsigned short`.

```
int pi = 3.14159265;
long l = 12.0;
unsigned short sh = 788999.56;
cout << pi << " " << l << " " << sh << endl;

Изход:
3   12   65535
```

Съществува още един начин типът на данните да бъде преобразуван. Това може да стане чрез оператора за преобразуване на типове `typecast`. Чрез него можем явно да укажем преобразуването. В този случай компилаторът не извежда предупреждение. Операторът за преобразуване на типовете се използва по следния начин:

```
double d = 4.5;
int x = 3 + (int)d;//x: 7
```

Бинарните аритметични операции за реални числа в езика са дефинирани както в математиката. При операции между цели числа те работят по същия начин с изключение на делението. Когато делението се осъществява между цели числа се нарича *целочислено деление*. Резултатът от целочисленото деление е отново цяло число, т.е. само цялата част от реалния резултат. Ето няколко примера:

```
4/2=2;      5/2=2;      11/4=2;     -9/2=-4;
```

Съществува още едно деление между цели числа и то се нарича *целочислено деление с остатък* и още *деление по модул*. То е

дефинирано да взема остатъчната част от целочисленото деление. Пресмята се по следния начин:

```
11 % 4= 3, защото
11 / 4= 2 проверка-> 2*4=8 остатък-> 11-8=3
```

Ето няколко примера:

```
8%2=0;      9%2=1;      5%10=5
```

Когато в израз е записано число без десетична точка (например 12), то се интерпретира като цяло, а когато е записано с десетична точка – като реално (например 12.0).

Целочислените деления са изключително полезни в програмирането. В часовете за упражнения ще бъдат предложени елегантни решения на задачи чрез използване на целочислените деления.

Унарните операции се прилагат върху променливи от всички аритметични типове данни. За променливи от типа `bool` те са дефинирани да не превъртат интервала [0;1]. Операциите могат да бъдат записвани и от двете страни на променливите, като това в изрази има различни значения. Ако в израз някоя от унарните операции `--` и ++ е написана преди променлива, то тя се изпълнява с най-висок приоритет (първа), т.е. променливата участва в израза с новата си стойност. Ако някоя от тези операции е написана след променлива, то тя се изпълнява с най-нисък приоритет (последна), т.е. променливата участва в израза със старата си стойност. Ето един пример:

```
char c = 96; int i = 0; double d = 1;
++c; i++; d--;     //c=97,  i=1,  d=0
double g = c * ++d - i--;
cout << g << " " << i << " " << d << endl;

Резултат:
96  0   1
```

Употребата на унарните операции в изрази повишава тяхната логическа сложност и не оптимизира изчислението. Поради това

влагането им в изрази не се препоръчва. Ето как може да се запише изразът от предходния пример:

```
char c = 96; int i = 0; double d = 1;
c++; i++; d--;    //c=97,  i=1,  d=0
d++;
double g = c * d - i;
i--;
cout << g << " " << i << " " << d << endl;
```

Глава IV. Оператори за сравнения. Логически операции. Условен оператор „?   :”. Побитови операции. Вградени функции

4.1. Оператори за сравнения.

В езиците за програмиране можем да описваме сравнения между стойностите на променливи, константи и изрази. В C++ това е възможно чрез операторите: > (по-голямо), >= (по-голямо или равно), == (равно), < (по-малко), <= (по-малко или равно), != (различно). Резултатът от всяка една от тях е една от целочислените стойности 0 и 1, а типът на данните за тяхното представяне е `bool`. За стойността 0 е дефинирана константата `false`, която означава „неистина” или „лъжа”, т.е. твърдението, което се прави чрез оператора за сравнение, не е вярно. За стойността 1 е дефинирана константата `true`, която означава „истина”, т.е. твърдението, което се прави чрез оператора за сравнение, е вярно. Ето един пример:

```
bool b = false, b1 = 5 > 10, b2 = 5 < 10;
cout << b << " " << b1 << " "  << b2 << endl;

Изход:
0   0   1
```

Булевите стойности 0 и 1 (false и true) наричаме *логически*.

4.2. Логически операции.

Освен с логически стойности в езиците за програмиране можем да работим и с *логически операции* [4]. Това са: *конюнкция* (логическо „И”), *дизюнкция* (логическо „ИЛИ”) и *отрицание* (логическо „НЕ”). В езика C++ тези операции се осъществяват съответно чрез операторите: `and (&&)`, `or (||)` и `not (!)`. В таблица 4.1 е дадена семантиката на `and` и `or`, които се прилагат бинарно. Операцията отрицание се прилага унарно и е дефинирана така: 1=!0 и 0=!1;

Таблица 4.1. Семантика на and и or.

a	b	and (&&)	or (||)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Няколко примера за употребата на and и or са дадени в следния програмен текст:

```
int a = 5, b = 10, c = 15;
bool b = a < b  and a < c;
bool h = a < b or c < b;
bool g = true;
cout << b << "  " << h << " " << not g << endl;

Изход:
1   1   0
```

При съставянето на логически изрази трябва да знаем и законите (на Аугустус де Морган) за отрицание на конюнкция и отрицание на дизюнкция:

```
not (a and b)      ⇔    not a or not b
not (a or b)       ⇔    not a and not b
```

4.3. Условен оператор „? :”.

Условните оператори в езиците за програмиране предоставят възможности за разклонения в програмите. Една програма, в която има условен оператор, може да продължи по два различни начина в зависимост от конкретно условие. Тук ще се спрем само на краткия условен оператор в C++. Ето неговия синтаксис:

```
<expression> ? <expression1> : <expression2>
```

Този условен оператор започва с израз, който трябва да връща или е стойност от скаларен тип. Тази стойност се преобразува до логическа (false или true). Ако стойността е равна на 0, тогава тя означава логическата стойност false, а ако тя е различна от 0, означава логическата стойност true. Ако логическата стойност на <expression> е true, тогава се изпълнява <expression1>, а в противен случай (логическата стойност на <expression> е false) се изпълнява <expression2>. Ето един пример:

```
(a >= 0) ? cout << "a>=0\n" : cout << "a<0\n";
```

Този оператор може да се поставя от лявата страна на оператора за присвояване на стойности. В този случай <expression1> и <expression2> трябва да бъдат променливи и целият оператор да бъде поставен в скоби. Ако логическата стойност на <expression> е true, тогава променливата, поставена на мястото на <expression1>, приема стойността вдясно от оператора =, а в противен случай променливата, поставена на мястото на <expression2>, приема стойността вдясно от оператора =. Ето един пример:

```
( a > b ? a : b ) = max;
```

4.4. Побитови операции.

Побитовите операции (от англ. bitwise operators) се извършват върху битовете на числовите двоични стойности на променливи и константи, без да се прилага пренос към старши разряди. Това са: „побитово И” (&) – аналог на and; „побитово ИЛИ” (|) – аналог на or; „побитово изключващо ИЛИ” (^ или xor); „побитово отрицание” (~) – аналог на not; „побитово преместване наляво” (<<); „побитово преместване надясно” (>>).

Таблица 4.2. Семантика на побитовите операции.

Оператор	Значение
&	побитово И
|	побитово ИЛИ
^	побитово изключващо ИЛИ

~	побитово отрицание
<<	побитово преместване наляво
>>	побитово преместване надясно

Побитовите операции „И”, „ИЛИ” и „изключващи ИЛИ” са бинарни, а „отрицание”, „преместване наляво” и „преместване надясно” са унарни. Побитовата операция „изключващо ИЛИ” е дефинирана да връща: 1 когато двата бита са с различни стойности и 0 когато двата бита са с еднакви стойности. В таблица 4.2 са дадени операторите, които се използват за побитовите операции.

Примери:

a	0	0	0	1	1	1	0	1
b	1	1	1	0	1	1	0	1
a & b	0	0	0	0	1	1	0	1

a	0	0	0	1	1	1	0	1
b	1	1	1	0	1	1	0	1
a	b	1	1	1	1	1	1	0	1

a	0	0	0	1	1	1	0	1
b	1	1	1	0	1	1	0	1
a ^ b	1	1	1	1	0	0	0	0

a	0	0	0	1	1	1	0	1
~ a	1	1	1	0	0	0	1	0

a	0	0	0	1	1	1	0	1
a << 2	0	1	1	1	0	1	0	0

a	0	0	0	1	1	1	0	1
a >> 3	0	0	0	0	0	0	1	1

Побитовата операция „И” може да се приложи за определяне на онези битове в операнда, където се намират единици чрез използване на маска от единици или намиране на битовете, които съдържат нули, при използване на маска от нули и побитово „ИЛИ”. Тези операции, заедно с „изключващо ИЛИ” намират приложение при създаване на програми за управление на специализирани периферни устройства, при предаване и криптиране на данни.

Побитовото преместване наляво премества битовете наляво с толкова позиции, колкото са указани със стойност след оператора,

като празните позиции вдясно се запълват с нули. Побитовото преместване надясно премества битовете надясно с толкова позиции, колкото са указани със стойност след оператора, като празните позиции вляво се запълват с нули.

Побитово преместване наляво на един бит по същество означава умножение с 2, два бита – умножение с 4 и т.н. Преместването надясно с всеки един бит дели числото на две. Това са изключително бързи операции на ниско ниво, което следва да се има предвид при разработване на различни програми.

Много често в програми се налага размяната на стойностите на две променливи. Обичайно това се прави с помощта на трета. Нека да предположим, че трябва да разменим стойностите на променливите a и b. Тогава можем да дефинираме нова променлива x, която да е от същия тип на данни, от който са a и b. Размяната можем да направим по следния начин:

```
x = a;
a = b;
b = x;
```

Размяната можем да направим и без да използваме трета променлива, а с помощта на операциите събиране и изваждане или с помощта на „побитово изключващо ИЛИ”.
С операциите събиране и изваждане:

```
a = a + b;
b = a - b;
a = a - b;
```

В първия вариант се печели време, защото се извършват само три преместавания в паметта (преместването е една от най-бързите операции), но е необходима допълнителна памет – още една клетка за междинната променлива x.

Във втория случай обратно: печели се памет, защото не се въвежда нова променлива, която някъде да се съхранява, но времето за изпълнение е значително по-голямо – необходими са едно събиране и две изваждания (събирането е поне 3-4 пъти по-бавна операция от преместването).

С помощта на „побитово изключващо ИЛИ”:

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

Това може да се запише и така:

```
a = a ^ ( b = b ^ ( a = a ^ b ) );
```

или по този начин:

```
a ^= b ^= a ^= b;
```

4.5. Оператори за присвояване на стойности.

В C++ освен със стандартния оператор за присвояване на стойности на променливи „=”, разполагаме и с няколко допълнителни. Това са: *=, /=, %=, +=, -=, &=, ^=, |=. Те служат за осъществяване на аритметични действия и побитови операции, които се прилагат към стойността на променливата, стояща вляво от съответния оператор. Например, програмният текст

```
int a = 0, b = 3;
a += 2;  b *= 3;  // a <- 2; b <- 9;
```

повишава стойността на a с 2 и умножава стойността на b с 3. Това е еквивалентно на:

```
int a = 0, b = 3;
a = a + 2; b = b * 3;
```

Операторът за присвояване на стойности на променливи може да бъде използван и в изрази по следния начин:

```
int a = b = 0, c = 10;
a = 2 + ( b = 3 + c );
```

В този израз променливата b ще приеме стойност 13 и след това променливата a ще приеме стойност 15.

4.6. Приоритети на операциите.

За операциите в езика C++ са дефинирани приоритети. Това означава, че операциите не винаги се изпълняват в последователността, в която са написани. В таблица 4.3 са дадени всички оператори в C++, подредени в низходящ ред по приоритет на операциите, които те извършват.

Таблица 4.3. Приоритети на операциите в C++.

<table>
<tr><th>Номер</th><th>Оператори</th><th>Значение</th></tr>
<tr><td>1</td><td>::</td><td>принадлежност</td></tr>
<tr><td>2</td><td>() [] -> .</td><td></td></tr>
<tr><td>3</td><td>sizeof new delete
& * -- ++ - + ~
!</td><td>унарни</td></tr>
<tr><td>4</td><td>(typecast)</td><td>преобразуване на типове</td></tr>
<tr><td>5</td><td>.* ->*</td><td></td></tr>
<tr><td>6</td><td>/ * %</td><td>бинарни</td></tr>
<tr><td>7</td><td>+ -</td><td>бинарни</td></tr>
<tr><td>8</td><td>>> <<</td><td>побитови</td></tr>
<tr><td>9</td><td>> >= < <=</td><td>сравнения</td></tr>
<tr><td>10</td><td>== !=</td><td>сравнения</td></tr>
<tr><td>11</td><td>&</td><td>побитово „И”</td></tr>
<tr><td>12</td><td>^</td><td>побитово „изкл. ИЛИ”</td></tr>
<tr><td>13</td><td>|</td><td>побитово „ИЛИ”</td></tr>
<tr><td>14</td><td>&&</td><td>„И”</td></tr>
<tr><td>15</td><td>||</td><td>„ИЛИ”</td></tr>
<tr><td>16</td><td>?:</td><td>условен оператор</td></tr>
<tr><td>17</td><td>= *= /= %= += -=
&= ^= |=</td><td>присвоявания</td></tr>
<tr><td>18</td><td>,</td><td>разделител на аргументи</td></tr>
</table>

4.7. Вградени функции.

В езика C++ са реализирани много функции, които са направени да са част от него, така че програмистите да могат да се възползват от тях. Наричат се *вградени*. Всички вградени функции, съдържания на хедър файлове и други ресурси за езика C++ могат да бъдат видени в C++ Reference Pages, като например: http://www.cplusplus.com/reference/.

4.7.1. Математически функции.

В програмите много често се налага употребата на математически функции. Това са тригонометрични функции, повдигане на степен, логаритми и др. В езика C++ те са реализирани във вградената библиотека `math`. Декларациите на функциите се намират в хедъра `math.h`, от което следва, че той трябва да бъде вмъкнат в програмата чрез директивата `#include`. Всяка функция от тази библиотека се извиква със своето име и параметри, поставени в скоби. След нейното изпълнение, тя обикновено връща стойност, която е изчислила. Параметрите могат да бъдат константи, променливи и изрази. Например, функцията `sin` трябва да бъде извикана с нейното име и веднага след него стойност, поставена в овални скоби и указваща ъгъл в радиани. Ето един пример:

```
cout << sin( 3.14159265 ) << endl;
```

Функцията `sin` очаква стойността, записана в скобите, да бъде в един от типовете `float`, `double` или `long double`, като изчислява и връща стойност в избрания тип. Една от декларациите на функцията изглежда така:

```
double sin( double );
```

В началото се записва типът на върната стойност на функцията, след това нейното име и в скоби типът на параметъра.

За нас е важно да знаем декларациите на функциите и техните значения (описания, от англ. descriptions), но не и техните дефиниции (как точно изчисляват стойността, която връщат). В таблица 4.4 са дадени някои от по-често използваните функции от библиотеката `math`. Всички функции от библиотеката могат да бъдат видени в местата с документации за езика като http://www.cplusplus.com, http://www.en.cppreference.com и др.

Таблица 4.4. Функции от библиотеката `math`.

Декларация на функция	**Значение на функция**
sin	Изчислява синуса на ъгъл, подаден в радиани.
cos	Изчислява косинуса на ъгъл, подаден в

	радиани.
tan	Изчислява тангенса на ъгъл, подаден в радиани.
log	Изчислява натурален логаритъм.
log10	Изчислява логаритъм при основа 10.
ceil	Закръгля подадената стойност нагоре. 3.0 ← ceil(2.3); -2.0 ← ceil(-2.3);
floor	Закръгля подадената стойност надолу. 2.0 ← floor(2.3); -3.0 ← ceil(-2.3);
round	Закръгля подадената стойност до най-близкото цяло число. 4.0 ← round(4.3); 5.0 ← round(4.5); -5.0 ← round(-4.5);
trunc	Връща цялата част от реално (дробно) число. 4.0 ← round(4.8);
abs	Връща абсолютна стойност.
sqrt	Изчислява корен квадратен.
pow	x^y ← pow(x, y)

4.7.2. Функция `exit()`.

Функцията `exit( int )` спира работата на програмата като връща код на операционната система. Код 0 означава, че програмата е приключила своята работа нормално, а код различен от 0 означава, че се е случила грешка или се е появил неочакван резултат. Дефиницията на функцията се намира в библиотеката `cstdlib`.

4.7.3. Функция `rand()`.

Функцията `rand()` (от `cstdlib`)връща цяло случайно число в интервала [0, `RAND_MAX`], `RAND_MAX=32767`. Изчисляването на това число става на базата на входна стойност, която се нарича *инициализатор* на генератора на случайни числа. Затова генерираното число по-често се нарича *псевдослучайно*. Инициализаторът се задава с функцията `srand(`*n*`)`. След това, при една и съща стойност на *n*, `rand` ще генерира една и съща последователност от псевдослучайни числа при всяко свое изпълнение. Един от начините да направим `rand` да генерира всеки път различна последователност от числа е като в `srand` зададем

броя на секундите от 01.01.1970 до момента на включването на компютъра. Това се прави така:

```
srand( time( NULL ) );
```

След като сме извикали `srand`, можем да генерираме псевдо-случайно число така:

```
unsigned j = rand();
```

Понякога в програмите се налага генерирането не само на нула и положителни цели псевдослучайни числа, а и на отрицателни и реални, а понякога и в различен диапазон от заложения за `rand`. Ето как можем да генерираме отрицателни, нула и положителни числа:

```
int j = rand() - RAND_MAX / 2;
```

Така се генерира цяло число в интервала `[-RAND_MAX/2, RAND_MAX/2]`.

Генерирането на реално число се демонстрира в цялостна програма:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    srand( time( NULL ) );
    double j = (double)( rand() - RAND_MAX/2 )
                   / rand() + rand();
    cout << j << endl;
    return 0;
}
```

Генериране на цяло число в интервал [0, m], където m<=RAND_MAX, може да се направи като числото, генерирано от `rand`, се раздели с целочислено деление с остатък на m+1.

```
int j = rand() % (m+1);
```

Това е така, защото при разделянето по модул на което и да е цяло число на цялото положително число n, се получава число в интервала [0, n-1]. А ето и генериране на цяло число в интервала [a, b], като a<b не е задължително:

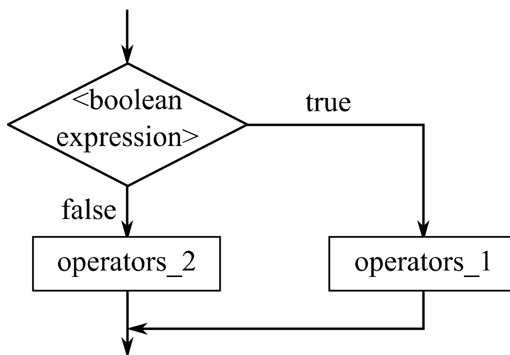
```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;
int main()
{
    srand( time( NULL ) );
    int a = 5, b = -10;
    int j = min(a,b) + rand() % abs(b-a);
    cout << j << endl;
    return 0;
}
```

Глава V. Оператори за разклонения

Много често в програмите се налага да бъдат реализирани разклонения. Разклоненията представляват пътища, по които програмата може да премине, като се изпълняват различни групи от оператори [4]. В C++ разклоненията се осъществяват с операторите `if else` и `switch`.

5.1. Условен оператор `if else`.

Операторът `if else` реализира разклонения от вида, даден в следната блок схема (фиг. 5.1):



Фиг. 5.1. Блок схема на оператора `if else`.

Блок схемата описва ситуация с две възможни състояния. Изразът <boolean expression> представлява логическа стойност 0 (false) или 1 (true). В зависимост от стойността на израза <boolean expression> в програмата се изпълнява една от двете групи оператори <operators_1> или <operators_2>. След изпълнението на една от групите, програмата продължава с изпълнението на следващите оператори.

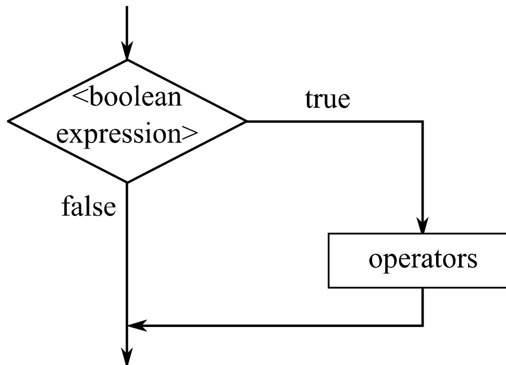
Синтаксисът в C++ на условния оператор е следният:

```
if( <expression> ) { operators_1 }
else { operators_2 }
```

```
и
if( <expression> ) operator_1
else operator_2
```

Изразът <expression> може да бъде от всякакъв скаларен тип данни, като стойността му определя логическа стойност 0 (false) или 1 (true). Ако стойността му е 0, то тя се преобразува до логическа 0 (false), а ако стойността му е различна от 0, то тя се преобразува до логическа 1 (true).

Най-естествено, разбира се, е изразът <expression> да бъде логически (булев). В този случай той се нарича *условие*. Ако oparators_1 или oparators_2 включват само по един оператор, тогава фигурните скоби могат да не се пишат.



Фиг. 5.2. Блок схема на съкратената форма на оператора `if else`.

Условният оператор може да съществува и в съкратена форма. Той реализира разклонение от вида, даден с блок схема на фигура 5.2. Съкратената форма на условния оператор има следният синтаксис:

```
if( <expression> ) { operators }
и
if( <expression> ) operator
```

Нека да разгледаме конкретна задача, чрез която да демонстрираме употребата на условния оператор.

Задача 4.1. Да се въведат три реални числа от клавиатурата. Програмата да извежда най-голямото от тях.

Тази задача може да бъде написана, следвайки различни разсъждения. Ще бъдат реализирани три от тях.

Решение 1: Дефинират се трите променливи `a`, `b` и `c`. Ако `a > b` и `a > c`, то тогава най-голямото число от трите е `a`. Аналогично разсъждаваме за `b` и `c`. Това можем да напишем така:

```
#include <iostream>
using namespace std;
int main()
{
   double a=0.0, b=0.0, c=0.0;
   cout << "Enter a, b and c: ";
   cin >> a >> b >> c;
   if( a > b and a > c )
           cout << "max: " << a << endl;
   if( b > a and b > c )
           cout << "max: " << b << endl;
   if( c > a and c > b )
           cout << "max: " << c << endl;
   return 0;
}
```

В този случай използваме само съкратената форма на условния оператор.

Решение 2: Ако `a > b`, тогава `b` отпада от „борбата" за най-голямо число. Остава да сравняваме `a` и `c` като ако `a > c`, тогава `a` е най-голямото, а в противен случай `c`. Но ако `a` не е по-голямо от `b`, тогава `a` отпада и трябва да продължим аналогично сравняване на `b` и `c`. Ето как се реализират тези разсъждения:

```
if( a > b )
      if( a > c )
            cout << "max: " << a << endl;
      else
            cout << "max: " << c << endl;
else
      if( b > c )
            cout << "max: " << b << endl;
      else
```

```
          cout << "max: " << c << endl;
```

В този пример се използва три пъти пълната форма на условен оператор. Два от операторите са вложени в един `if else`. Единият е в частта на `if`, а другият – в `else`. Фигурни скоби не се използват, тъй като в частите за оператори е поставен само по един оператор (друг условен оператор).

Решение 3: За откриването на най-голямото число от трите, можем да използваме една допълнителна променлива (`max`). Тя в началото може да присвои стойността на едното от трите числа. След което нейната стойност ще сравняваме последователно с останалите две числа. Ако открием число по-голямо от стойността в `max`, тогава `max` ще присвоява тази по-голяма стойност. Накрая в `max` ще е записано най-голямото число от трите. Ето реализацията:

```
max = a;
if( max < b ) max = b;
if( max < c ) max = c;
cout << "max: " << max << endl;
```

Влагането на по-дълъг израз в частта за условие в условен оператор допълнително натоварва смисъла на конструкцията. В този случай е удачно стойността, която изразът изчислява, да бъде предварително присвоена на променлива. След това тази променлива да се постави в частта за условие. Ето един пример:

```
bool bEx = a > b and a > c;
if( bEx ) cout << "max: " << a << endl;
```

В случай, когато изразът в <expression> е от другите скаларни типове данни, а не от `bool`, получената стойност се преобразува до логическа. Това е показано в следния програмен текст:

```
double d = price1 - price2;
if( d )
  cout<<"Има разлика между двете цени!\n";
else
  cout<<"Няма разлика между двете цени.\n";
```

В този случай, ако стойността на променливата `d` е `0.0`, се преобразува до логическата `0`, а ако е отрицателна или положителна, се преобразува до логическата `1`.

Нека да разгледаме реализацията на още една задача, която отлично илюстрира преимуществата на условния оператор.

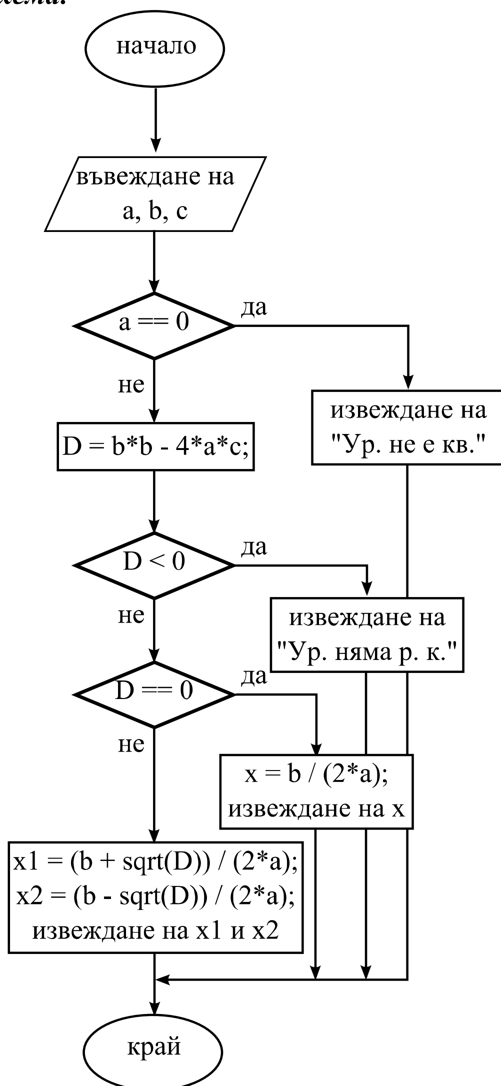
Задача 4.2. Да се разработи програма, която намира и извежда реалните корени на квадратно уравнение, чиито коефициенти се въвеждат от клавиатурата.

Анализ на задача 4.2: Нека трите коефициента, които ще се въвеждат от клавиатурата да означим с `a`, `b` и `c`. За да съществува квадратно уравнение, трябва да е изпълнено условието $a \neq 0$. Ако $a = 0$, то уравнението не е квадратно. Реални корени ще могат да бъдат намирани, ако за дискриминантата $D = b^2 - 4ac$ е изпълнено условието $D \geq 0$.

Алгоритъм:

Въвеждане на a, b, c;
Ако a=0 →
    извеждане на „Ур. не е квадратно”;
В противен случай →
    *D := b*b – 4*a*c;*
    Ако D<0 →
        извеждане на „Ур. няма р. к.”;
    В противен случай →
        Ако D=0 →
            *x := b / (2*a);*
            извеждане на x;
        В противен случай →
            *x1 := (b + sqrt(D)) / (2*a);*
            *x2 := (b – sqrt(D)) / (2*a);*
            извеждане на x1 и x2;
Край

Блок схема:



Реализация на задача 4.2:

```
#include <iostream>
#include <math.h>
using namespace std;
```

```
int main()
{
    double a, b, c;
    cout << "Въведете a, b, c:";
    cin >> a >> b >> c;
    if( a == 0 )
        cout << "Уравнението не е квадратно!\n";
    else {
        double D = b*b - 4*a*c;
        if( D < 0 )
            cout << "Уравнението няма р. к.\n";
        else
            if( D == 0 ) {
                double x = b / (2*a);
                cout << "x = " << x << endl;
            }
            else {
                double x1, x2;
                x1 = (b + sqrt( D )) / (2*a);
                x2 = (b - sqrt( D )) / (2*a);
                cout << "x1 = " << x1 << endl;
                cout << "x2 = " << x2 << endl;
            }
    }

    return 0;
}
```

5.2. Оператор `switch`.

Операторът `switch` има следният синтаксис:

```
switch( <expression> ) {
case <value1>: <operators1>
case <value2>: <operators2>
. . .
case <valueN>: <operatorsN>
default: <operators>
}
```

Изразът <expression> може да бъде от целочислен тип данни. Стойностите <value1>, <value2> и т.н. до <valueN> трябва да бъдат целочислени константи. Броят на константите е неограничен. Операторът търси съвпадения на стойността на <expression> с някоя от стойностите, поставени след всяка ключова дума `case`. Ако такова съвпадение е намерено, тогава се изпълняват операторите след двоеточието на случая, в който е съвпадението, и всички останали оператори надолу, но без операторите на `default`. Секцията `default` не е задължителна, но ако фигурира, операторите след `default` се изпълняват само ако не са намерени съвпадения.

Действието на `switch` може да бъде прекъснато преждевременно с оператора `break`.

Употребата на switch ще бъде демонстрирана с реален пример.

Задача 4.3: Да се напише програмен текст, който въвежда цяло положително число от 1 до 7 от клавиатурата, след което извежда съответстващия му ден от седмицата.

Реализация на задача 4.3:

```
unsigned short d = 0;
cout << "Enter a number between 1 and 7: ";
cin >> d;
switch( d ) {
case 1: cout << "Monday\n"; break;
case 2: cout << "Tuesday\n"; break;
case 3: cout << "Wednesday\n"; break;
case 4: cout << "Thursday\n"; break;
case 5: cout << "Friday\n"; break;
case 6: cout << "Saturday\n"; break;
case 7: cout << "Sunday\n"; break;
default: cout<<"The number is not in [1,7]!\n";
}
```

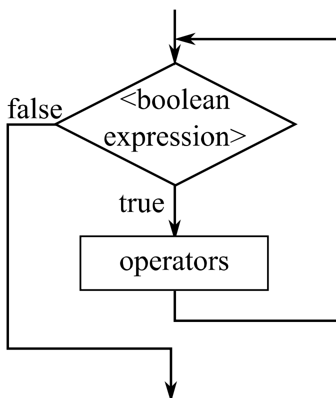
След въвеждането на числото от клавиатурата се търси съвпадение на `d` с числата от `1` до `7`. Ако такова бъде открито, се извежда съответстващият му ден от седмицата. За да не продължи изпълнението на следващите оператори, в `switch` се използва операторът `break` в края на всеки случай. Ако съвпадение не бъде открито, се извежда съобщението, гласящо, че въведеното число не е между `1` и `7`.

Глава VI. Оператори `while, do while и for`

Чрез операторите while, do while и for се реализират циклични структури (цикли) в C++. Много често в програмите се налага многократното изпълнение на определени програмни текстове. Това се прави като даденият програмен текст се постави в един от операторите за реализиране на цикли.

6.1. Цикъл `while`.

Операторът while реализира циклични структури от вида, даден в следната блок схема (фиг. 6.1):



Фиг. 6.1. Блок схема на циклична структура, реализираща се чрез while.

Синтаксисът на оператор while е следният:

```
while( <expression> ) { operators }
и
while( <expression> ) operator
```

Изразът <expression> може да бъде от всякакъв скаларен тип данни, като стойността му определя логическа 0 (false) или логическа 1 (true). Ако стойността му е 0, то тя се преобразува до логическа 0

(false), а ако стойността му е различна от 0, то тя се преобразува до логическа 1 (true).

Най-естествено е изразът <expression> да бъде логически (булев). В този случай той се нарича *условие*. Ако частта за оператори включва само един оператор, тогава фигурните скоби може да не се пишат.

Семантиката на оператора е следната: Докато условието е логическа 1 (или стойността на <expression> е преобразувана до логическа 1), изпълнявай операторите (operators). Това означава, че първо се проверява условието и ако то е истина, се изпълняват операторите, след което отново се проверява условието, за да се прецени дали да се изпълнят отново операторите. Ако условието не е истина, а лъжа, действието на оператора се прекратява. Едно изпълнение на операторите в цикъл се нарича *итерация*.

Тъй като операторът се изпълнява само при истинно условие, следва, че ако то не е такова в началото, операторите (operators) няма да се изпълнят нито веднъж.

Секцията, съдържаща условието в циклите, може да бъде оставена празна. Това се интерпретира като логическа стойност 1. Това означава, че ако цикъл не съдържа условие, то той ще се изпълнява безкрайно. Нарича се *безкраен цикъл*.

Действието на оператора `while` ще бъде демонстрирано със задача, която ще реализираме след това и с другите оператори за цикъл.

Задача 5.1. Да се напише програмен текст, който въвежда цяло положително число от клавиатурата и извежда всички цели числа от 1 до въведеното включително.

Решение на задача 5.1 с `while`: За да не се усложнява примерът, ще предполагаме, че потребителят въвежда винаги цяло положително число. Тогава то ще бъде със стойност 1 или по-голямо. Нека да съхраняваме тази стойност в променлива `n`. Тогава, започвайки от 1, трябва да извеждаме числата `1, 2, ..., n`. Стойностите, които ще извеждаме, нека да съхраняваме в променлива `i`. Нейната стойност първоначално може да бъде 1. Извеждането на целите числа от 1 до `n` трябва да се направи циклично, чрез натрупване на стойностите последователно в `i`. Следователно ще използваме цикъла `while`. Неговото условие може да бъде `i <= n`. Това означава, че докато `i` е по-малко или равно на

n, извеждането на стойността на i и инкрементирането ѝ, ще продължават. Когато стойността на i стане по-голяма от n, изпълнението на цикъла ще спре и програмата ще продължи нататък. Ето и програмния текст:

```
unsigned i = 1, n = 0;
cin >> n;
while( i <= n ) {
      cout << i << endl;
      i++;
}
```

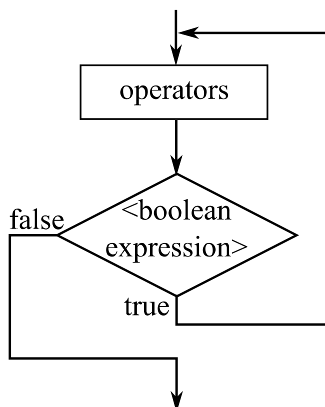
Проиграване на работата на цикъла:

while(i <= n)				
i	1	2	...	n
cout << i	1	2	...	n
i++	2	3	...	n+1

Ако въведеното число от клавиатурата е 0 или по-малко от 0, то цикълът няма да се изпълни нито веднъж. Ако въведеното число е 1, тогава цикълът ще се изпълни само веднъж. Във всички случаи, когато n >= 1, цикълът ще се изпълнява n пъти. Изпълнението на цикъла while винаги ще завършва със стойност на i = n+1.

6.2. Цикъл do while.

Операторът do while реализира циклични структури от вида, даден в следната блок схема (фиг. 6.2):



Фиг. 6.2. Блок схема на циклична структура, реализираща се чрез `do while`.

Синтаксисът на оператор `do while` е следният:

```
do { operators } while( <expression> );
и
do operator while( <expression> );
```

Изразът <expression> има същото значение както при цикъла `while`.

Семантиката на оператора е следната: Изпълняват се операторите (operators), докато условието е логическа 1 (или стойността на <expression> е преобразувана до логическа 1). Това означава, че първо се изпълняват операторите (operators), а след това се проверява условието и ако то е истина, отново се изпълняват операторите. Ако условието не е истина, а лъжа, действието на оператора се преустановява.

Операторът `do while` се изпълнява винаги поне веднъж, тъй като в него първо се изпълняват операторите и след това се проверява условието.

Действието на оператора `do while` ще бъде демонстрирано чрез реализацията на задача 5.1 с този цикъл.

Решение на задача 5.1 с `do while`: Тази реализация запазва максимално от предходния програмен текст като използва цикъла `do while`.

```
unsigned i = 1, n = 0;
```

```
cin >> n;
do {
      cout << i << endl;
      i++;
} while( i <= n );
```

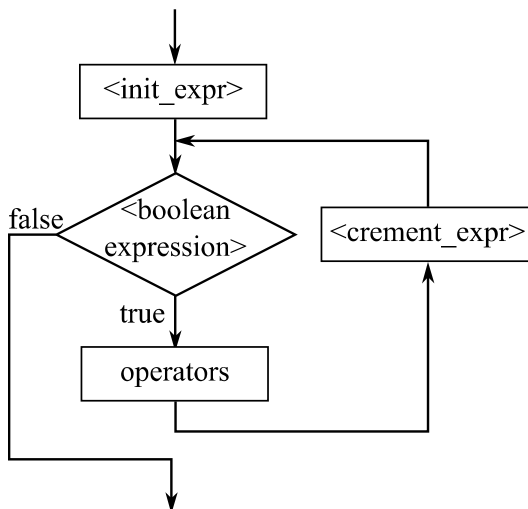
От реализацията се вижда, че каквато и стойност да бъде въведена за `n`, цикълът винаги ще извежда поне 1.

Проиграване на работата на цикъла:

do … while(i <= n)				
i	1	2	...	n
cout << i	1	2	...	n
i++	2	3	...	n+1

6.3. Цикъл `for`.

Операторът `for` реализира циклични структури от вида, даден в следната блок схема (фиг. 6.3):



Фиг. 6.3. Блок схема на циклична структура, реализираща се чрез `for`.

Синтаксисът на оператор `for` е следният:

```
for( <init_expr>; <expression>; <crement_expr> )
```

```
{ operators }
и
for( <init_expr>; <expression>; <crement_expr> )
operator
```

В секцията <init_expr> се задават начални стойности на променливи (обикновено участващи в условието). Секцията <expression> има същото значение както при цикъла `while`. В секцията <crement_expr> се правят корекции на стойностите на променливите.

Семантиката на оператора е следната: Изпълнява се кодът от секцията <init_expr>. Това се прави еднократно. След това се проверява стойността на <expression>. Изразът <expression> има същото значение както при цикъла `while`. Ако стойността на условието е логическата 1, следва изпълнение на операторите (operators). След това се изпълнява кодът в секцията <crement_expr>. А при стойност на условието логическа 0, цикълът преустановява своята работа.

Тъй като операторът се изпълнява само при истинно условие, следва, че ако то не е такова в началото, операторите (operators) няма да се изпълнят нито веднъж.

Действието на оператор `for` ще бъде демонстрирано чрез реализацията на задача 5.1 с този цикъл. Тя запазва максимално от предходния програмен текст като използва цикъла `for`.

Решение на задача 5.1 с `for`:

```
unsigned n = 0;
cin >> n;
for( unsigned i = 1; i <= n; i++ )
    cout << i << endl;
```

Проиграване на работата на цикъла:

for(i = 1; i <= n; i++)				
i	1	2	...	n
cout << i	1	2	...	n
i++	2	3	...	n+1

Секциите на цикъла `for` могат да бъдат оставени празни. В този случай обаче знаците `';'`, трябва да се пишат. Ето един пример:

```
unsigned i = 1, n = 0;
cin >> n;
for( ; i <= n; ) {
      cout << i << endl;
      i++;
}
```

Ако секцията <expression> се остави празна, цикълът става безкраен. Безкрайни цикли се използват рядко. Понякога безкраен цикъл може да се получи неумишлено – при условие, което никога не може да стане „лъжа”, въпреки че очакването ни не е такова.

Изпълнението на всеки цикъл може да бъде прекратено чрез оператор `break`, поставен в частта за оператори. Последствията са, че цикълът се прекратява и програмата продължава нататък. Ето един пример:

```
for( i = 1; i <= n; i++ ){
      cout << i << endl;
      if( i >= m )
            break;
      m--;
}
```

В този програмен код се извежда стойността на `i` като на всяка итерация се инкрементира, а стойността на `m` се декрементира. Когато стойността на `i` стане по-голяма или равна на `m`, цикълът се прекратява от `break`.

Изпълнението на итерация на цикъл може да бъде прекратено чрез оператор `continue`, поставен в частта за оператори. Последствията са, че операторите, поставени след `continue`, няма да бъдат изпълнени в итерацията, но цикълът ще продължи своята работа с нова итерация. Употребата му е демонстрирана в програмен текст, който извежда четните числа от 1 до 100:

```
for( i = 1; i <= 100; i++ ) {
      if( i % 2 == 1 )
            continue;
      cout << i << endl;
}
```

6.4. Предусловие и постусловие.

Операторите за цикли се изпълняват в зависимост от условие и се делят условно на два вида: с *предусловие* (условието е поставено преди операторите за изпълнение) и с *постусловие* (условието е поставено след операторите за изпълнение). Операторите с предусловие са `while` и `for`, а с постусловие – `do while`. Най-характерната причина за това разделение е, че операторите с предусловие могат да не се изпълнят нито веднъж, а този с постусловие винаги се изпълнява поне веднъж.

Задача 5.2. Да се напише програма, която подканва потребителя да въведе курс на долара към лева. След това програмата да чете въведени стойности в долари и да ги превръща в лева като за сентинел (признак за край на въвеждането) се използва 0.

Анализ на задача 5.2: Задачата изисква сумите в долари да бъдат въвеждани циклично от клавиатурата. Това трябва да става до въвеждането на 0 т.е. докато стойността в долари е различна от 0. За да бъде обаче това проверено, трябва вече да имаме въведена стойност в долари. Следователно, трябва ни цикъл с постусловие. Такъв единствен е цикълът `do while`.

Реализация на задача 5.2:

```
#include <iostream>
using namespace std;

int main()
{
    double rate=0.0, dollars=0.0, leva=0.0;
    cout << "Enter the rate (1$ = ? leva): ";
    cin >> rate;
    do {
        cout << "Enter amount in dollars: ";
        cin >> dollars;
        leva = dollars * rate;
        cout<<dollars<<" $ = "<<leva<<" lv.\n";
    } while( dollars != 0.0 );
    return 0;
}
```

6.5. Оператор goto.

Операторът goto служи за преминаване от една точка в програмата (точката, в която е извикан) в друга точка, наречена *назначение*. Назначението се определя чрез *етикет*. Етикетът трябва да бъде валиден идентификатор, последван от двоеточие. Чрез goto биха могли да се осъществяват циклични структури. Това ще бъде демонстрирано чрез реализация на задача 5.1 с goto:

```
unsigned i = 1, n = 0;
cin >> n;
loop_lable:
if( i <= n ) {
      cout << i << endl;
      i++;
      goto loop_lable;
}
```

6.6. Вложени цикли.

При определени задачи се налага влагане на цикъл в цикъл. В такъв случай циклите се наричат *вложени*. Влагане на цикли ще бъде демонстрирано с реализацията на следващата задача.

Задача 5.3. Да се изведе фигурата от числа

1
1 2
1 2 3
. . . .
1 2 3 4 5 10

Анализ на задача 5.3: Трябва да бъдат изведени 10 реда като на първия ред се извежда 1, на втория – от 1 до 2, на третия – от 1 до 3 и т.н. Следователно на всеки ред трябва да се извеждат числата от 1 до номера на реда. Ако направим един цикъл, който да има десет итерация с индексна променлива i започваща от 1, то в него можем да вложим друг цикъл, който да се изпълнява за своя индексна променлива j за стойности от 1 до i, като извежда стойността на j и един интервал. В края на всяка итерация на първия цикъл трябва да се извежда знак за край на ред.

Реализация на задача 5.3:

```
for( int i = 1; i <= 10; i++ ) {
      for( int j = 1; j <= i; j++ )
            cout << j << " ";
      cout << endl;
}
```

Цикълът `for( int i = 1;`... извежда десетте реда с числа, а цикълът `for( int j = 1;`... извежда числата от 1 до `i` на `i`-тия ред. Последният споменат цикъл се извиква 10 пъти и всеки път се изпълнява `i` пъти.

Задача 5.4. Да се напише програма, която подканва потребителя да въведе естествено число и проверява за коректността на потребителския вход. Програмата да намира и извежда средното аритметично на цифрите на числото.

Анализ на задача 5.4: След въвеждането на число от клавиатурата трябва да се провери дали потребителският вход е коректен (дали входният поток е пропаднал или числото е отрицателно). Сумата и броят на цифрите на числото могат да се намерят като се извлече всяка цифра на числото. За целта трябва да се използва целочислено деление. Тъй като броят на цифрите на числото първоначално не е известен, извличането на цифрите трябва да стане от последната (най-дясната) до първата (най-лявата).

Реализация на задача 5.4:

```
#include <iostream>
using namespace std;

int main()
{
    long num = 0, x = 0, y = 0, Sum = 0, i = 0;
    cout<<"Enter a nonnegative integer number:";
    cin >> num;
    if( !cin or num < 0 ) {
        cout << "Incorrect input!\n";
    }
    else{
        x = num;
        while( x > 0 ){
            y=x%10;//извлича най-дясната цифра
```

```
            Sum += y;//прибавя извлечената цифра
            x=x/10;//премахва най-дясната цифра
            i++;   //преброява цифрата
        }
        cout <<"Average:"<<(double)Sum/i <<endl;
    }
    return 0;
}
```

Глава VII. Указатели. Масиви

7.1. Указатели.

Данните и функциите, с които се работи в една програма, се съхраняват в оперативната памет на компютъра [7]. Представителите на обектите в програмата са променливите и именуваните константи. Следователно, използвайки идентификаторите им, ние записваме и четем в и от паметта. Използвайки имената (идентификаторите) на функциите, ние можем да ги извикваме в нашите програми. Но как става връзката между идентификаторите и данните или функциите, които те представляват?

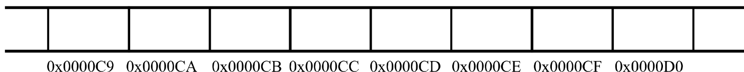
Всеки байт от оперативната памет е еднозначно определен от номер, наречен още *адрес*. Адресите обикновено се дават в шестнадесетични числа, които се записват с префикс 0x. Първият байт от паметта има адрес 0x000000, следващият 0x000001 и т.н. Двеста петдесет и шестият байт в паметта например има адрес 0x0000FF ($=255_{(10)}$).

В езика C++ ние можем да четем и записваме данни в и от паметта, използвайки адреси. Променливите, които можем да дефинираме, за да съхраняваме адреси в тях, се наричат *указатели*. Указателите се дефинират от съществуващ тип на данни (стандартен или дефиниран) чрез оператора `*`, поставен преди идентификатора на променливата по следния начин:

```
<type> * <name>;
```

Адресът на променлива или именувана константа се извлича чрез оператора `&`, поставен непосредствено пред идентификатора. Това връща адреса на първия байт от последователността байтове, които са необходими за представяне на данните в конкретния тип. Например, типът `double` представя числата в 8 байта. Ако в програма имаме променлива `d` от този тип с адрес 0x0000C9 ($=201_{(10)}$), то заеманите от нея байтове са от адрес 0x0000C9 до адрес 0x0000D0 ($=208_{(10)}$) включително (фиг. 7.1). Операторът `&` не

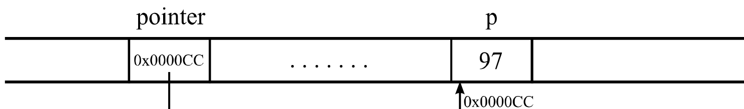
може да бъде прилаган върху неименувани константи и аритметични изрази като например: `&5`, `&(a-5)`.



Фиг. 7.1. Байтове в паметта и техните адреси.

Причината, указателите да са обвързани с типа на данните, е необходимостта да се знае колко байта от началния адрес и след това трябва да бъдат използвани (за запис или четене). За указател, на който вече е присвоен адрес, казваме, че той *сочи* към конкретен обект в паметта. Ето един пример, който демонстрира дефинирането на указател, сочещ към конкретен обект от паметта (фиг. 7.2):

```
unsigned char p = 97;
unsigned char *pointer = &p;
cout << hex << (int)pointer << endl;
```



Фиг. 7.2. Указател, сочещ към обект в паметта.

Стойността на обекта, към който сочи указателят, може да бъде извлечена чрез оператора `*`, поставен пред идентификатора на указателя. Предходният пример може да бъде продължен с извеждане на стойността на обекта, към който сочи указателят `pointer`:

```
cout << *pointer << endl;

Изход:
97
```

Ако указател има стойност 0 (може да се ползва и константата `NULL = 0`), това означава, че той не сочи към данни в

паметта. Указатели от един и същи тип могат да приемат своите стойности чрез оператор за присвояване, както е показано в следващия пример:

```
int *p1 = 0, *p2 = NULL;
int h = 10;
p1 = &h; p2 = p1;
cout << *p2 << endl;
```

Указателите могат да сочат само към обекти от типа на данни, от които са дефинирани, освен когато са дефинирани от тип `void`. Указатели от тип `void` могат да бъдат присвоявани на указатели от всякакви типове, но чрез оператора за преобразуване на типовете. Указатели от всякакви типове могат да бъдат присвоявани на указатели от тип `void` без преобразуване на типовете. Тези особености се демонстрират в следния пример:

```
void *v; int *p; int i = 0, j = 1;
v = &i; p = (int*)v;
cout << *p << endl;
p = &j; v = p;
cout << *(int*)v << endl;
```

Указатели могат да бъдат дефинирани да сочат към константи. Това става по следния начин:

```
const int ci = 3;
const int *q = &ci;
```

В този случай промяна на стойността на обекта, към който сочи указателят, е недопустима (~~`*q=5;`~~).

Указатели могат да бъдат дефинирани като константи и се наричат *указатели константи*. Това се прави по следния начин:

```
int k = 3;
int * const q = &k;
```

В този случай промяна на стойността на указателя не е допустима (~~`q=&i;`~~). Допуска се указатели, дефинирани да сочат към константи, да приемат адреси на променливи.

Указатели константи могат да бъдат дефинирани да сочат към константи. Това се прави по следния начин:

```
const int k = 3;
const int * const q = &k;
```

В този случай както промяната на стойността на указателя, така и промяната на стойността на обекта, към който сочи указателят, са недопустими.

Указатели могат да бъдат дефинирани да сочат към други указатели. Наричат се *указатели към указатели*. Ето един пример:

```
int k = 3;
int *q = &k;
int **g = &q;
int ***f = &g;
cout << ***f << endl;

Изход:
3
```

Операциите, които можем да прилагаме върху указатели са следните: =, <, <=, ==, >, >=, !=, -, +, --, ++. Употребата на оператора = вече беше изяснена. Сравненията се извършват като се сравняват адресите по стойност. Операторите -, +, -- и ++ се извършват по правилото на така наречената *адресна аритметика*: при изпълнение на операция за повишаване или намаляване на стойността на указател, неговият адрес съответно се увеличава или намалява с посочената стойност, умножена по броя на байтовете, които са необходими за представяне на данните в типа на указателя. Ако `p` е указател, дефиниран към тип `type` и насочен към адрес на обект в паметта, то правилото на адресната аритметика може да бъде записано така:

```
p+j → p+j*sizeof(type)
```

Например, ако имаме дефиницията `double *pd = &m;` и `pd` има стойност 0x0000C9 ($=201_{(10)}$), то изразът `pd++;` ще присвои на `pd`

стойност 0x0000D1 ($=209_{(10)}$), а изразът `pd = pd + 2;` ще присвои на `pd` стойност 0x0000D9 ($=217_{(10)}$).

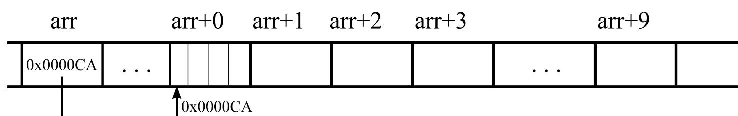
Адресната аритметика може да бъде прилагана ефективно върху обекти, които са разположени непосредствено един след друг в паметта, каквито са например масивите.

7.2. Масиви.

Масивите представляват множество от последователно наредени обекти (непосредствено един до друг) от един и същи тип данни в паметта [4]. Адресът на началния байт на тази последователност от обекти се съхранява в указател. Преминаването от един обект към друг се осъществява чрез адресна аритметика.

7.1.1. Едномерни масиви.

Дефинирането на масив (от англ. array) става като се зададе тип на обектите (елементите на масива), име на масива (идентификатор) и в правоъгълни скоби брой на елементите. Масивът в този вариант се нарича *едномерен*, тъй като представя едномерна структура от данни (фиг. 7.3). Името на масив представлява указател, който съдържа адреса на първия байт от първия елемент на масива.



Фиг. 7.3. Едномерен масив в паметта.

Нека дефинираме масив от тип `int` с 10 елемента:

```
const int N = 10;
int arr[ N ];
```

Достъп до елементите на масива се осъществява чрез адресна аритметика. Първият байт на първия елемент има адрес, записан в `arr` (също можем да го бележим като `arr+0`). Да предположим, че адресът е 0x0000CA. Следователно, първият елемент е разположен в байтовете с адреси от 0x0000CA до 0x0000CD (4 байта). Следващият елемент от масива започва от байт с адрес `arr+1`, чиято стойност ще

бъде 0x0000CE. Последният елемент от масива ще е разположен на адрес `arr+9` (или в общия случай `arr+N-1`). Числата от `0` до `N-1` се наричат *индекси* на елементите на масива. Индексирането на елементите трябва да се извършва много внимателно, тъй като лесно може да се излезе извън диапазона [0, N-1]. Излизането от диапазона (и в двете посоки) ще представлява работа с обекти от паметта, които не принадлежат на масива, като е възможно да съдържат данни на програмата или данни на други програми, работещи в момента.

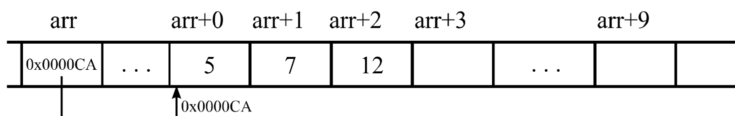
След дефинирането на масив в една програма, обикновено стойностите на неговите елементи са неопределени. Само някои компилатори извършват нулиране на елементите, ако масивите са от стандартните скаларни типове данни.

Задаването на стойности на елементите на масив става чрез задаване стойност на всеки елемент поотделно. Ето как можем да зададем стойности на първите 3 елемента от масива, който дефинирахме (фиг. 7.4):

```
*(arr + 0) = 5;
*(arr + 1) = 7;
*(arr + 2) = 12;
```

Това може да бъде записано и така:

```
arr[ 0 ] = 5;
arr[ 1 ] = 7;
arr[ 2 ] = 12;
```



Фиг. 7.4. Задаване на стойности на едномерен масив.

Често стойности на елементите на масив се задават чрез потоци. Нека да зададем стойности на елементите на масива от клавиатурата и след това да ги изведем в командния интерпретатор, разделени със запетаи:

```
for( int i = 0; i <= N-1; i++ ){
```

```
        cout << "arr[" << i << "]=";
        cin >> arr[ i ];
}
for( int i = 0; i <= N-1; i++ )
        cout << arr[ i ] << ", ";
```

Дефинирана е една целочислена променлива (`i`), която в контекста на работата с масив, можем да наричаме *индексна*, тъй като служи за индексирането на елементите му. Броят на елементите е N, следователно броят на итерациите на циклите трябва да бъде N, като индексната променлива започне от стойност 0 и след всяка итерация се инкрементира. Работата на цикъла продължава докато `i<=N-1`. В първия цикъл първо извеждаме съобщение за това, кой елемент ще се въвежда и след това го записваме в `arr[ i ]`. По друг начин казано: въвеждаме стойности за `arr[ i ]` за `i = 0,..., N-1`. Вторият цикъл извежда стойностите на елементите на масива, разделени със запетаи.

Извеждане на елементите на масива в обратен ред можем да направим така:

```
for( int i = 0; i <= N-1; i++ )
        cout << *( arr + N-1 - i ) << ", ";
                    arr[ N-1 – i ]
```

Задаването на стойности на елементите на масив може да бъде направено и в момента на дефинирането му по следния начин:

```
int a[ 5 ] = { 1, 7, 3, 12, 5 };
```

Броят на елементите в този случай може да не бъде указан явно:

```
int a[] = { 1, 7, 3, 12, 5 };
```

7.1.2. Многомерни масиви.

Чрез масивите могат да се представят многомерни структури от данни. *Многомерни масиви* се дефинират по същия начин както и едномерните, но с тази разлика, че броят на елементите по всяко измерение се задава в отделни правоъгълни скоби. Нека да предположим, че искаме да съхраним в паметта следната матрица от реални числа:

$$\begin{pmatrix} 3 & 2.1 & -1 & 5.5 & 3.14 \\ 4 & 5 & 7 & 33 & 55 \\ 0.11 & 7.66 & 3.4 & 8.5 & 22 \\ 33 & 55 & 44 & 22 & 11 \end{pmatrix}$$

Тази матрица се състои от 4 реда и 5 стълба. Можем да дефинираме масив, който да съхранява числата ред след ред, като ползваме два индекса – един за реда и един за стълба (индекса на елемента в реда). Такъв масив се нарича *двумерен*. Нека да дефинираме двумерен масив с елементи от тип `double`, който да може да съхрани числата от матрицата:

```
const int M = 4, N = 5;
double d[ M ][ N ];
for( int i = 0; i <= M-1; i++ )
      for( int j = 0; j <= N-1; j++ ){
            cout <<"d["<< i << "][" << j <<"]=";
            cin >> d[ i ][ j ];
      }
```

Броят на редовете в този двумерен масив е `M`, а броят на елементите във всеки ред е `N` (броя на стълбовете). Общият брой на елементите на масива е MxN. За обхождането на масива са необходими две индексни променливи `i` и `j`. Обхождането може да се извършва както по редове, така и по стълбове. В програмния текст е избрано обхождане по редове, т.е. чрез `i` се фиксира ред, след което чрез `j` се обхождат всички негови елементи, като се въвеждат стойности от клавиатурата. Извеждането на елементите на двумерния масив в текстовия интерфейс може да се направи така:

```
for( int i = 0; i <= M-1; i++ ) {
      for( int j = 0; j <= N-1; j++ ){
             cout<<setw(10)<<d[i][j];
      }
      cout << endl;
}
```

Инициализирането на стойностите на елементите на двумерен масив може да се направи още при дефинирането му по следния начин:

```
int a[][3]={       {1,2,3},
                   {4,5,6} };
```

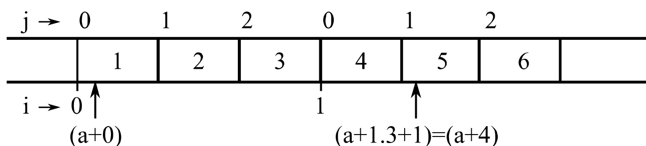
Двумерните масиви представят двумерни структури от данни, но оперативната памет е едномерна (линейна). Как тогава става представянето на данните на двумерните масиви в паметта? Редовете от елементи на двумерния масив се разполагат един след друг в паметта. Достигането на елемент се прави чрез адресна аритметика по правилото за намиране на адрес. Нека да вземем за пример последния дефиниран масив. Адрес на елемент с индекси `i` и `j` съответно за ред и стълб се получава така:

```
a + i.N + j, за i = 0,..,M-1 и j = 0,..,N-1
```

Този запис е еквивалентен на

```
a[ i ][ j ] , за i = 0,..,M-1 и j = 0,..,N-1
```

Ето един пример за намиране на адреса на елемент с индекси [1,1], т.е. `&a[1][1]`: a + 1.3 + 1 = a + 4. Това е демонстрирано на фигура 7.5.



Фиг. 7.5. Адресна аритметика за двумерен масив.

Възможно е дефинирането на масиви, които представят тримерни структури от данни. В този случай масивите се наричат *тримерни*. Ето един пример за дефинирането на тримерен масив с размерност 2x2x3:

```
int a[2][2][3]={ {{1,2,3}, {3,4,5} },
                { {6,7,8}, {4,7,9} }
```

```
};
```

Този масив се състои от два реда, два стълба и три елемента в дълбочина за всеки ред и стълб.

Възможно е дефинирането на масиви, които представят n-мерни структури от данни. В този случай масивите се наричат *n-мерни*.

Глава VIII. Динамична памет

Чрез променливите, константите и масивите се създават обекти в оперативната памет, които служат за съхраняване на данни. Паметта на тези обекти остава заета до момента на завършване на работата на функцията, в която са дефинирани, а понякога и до завършване на програмата [5].

При работа с голямо количество данни в една програма отделената памет за тях ще бъде също голяма. Но много рядко се налага работа с голямо количество данни в един момент. С цел оптимално използване на паметта е удачно в нея да бъдат записвани само данните, които трябва да се обработват в даден момент, а тези, които вече не са необходими, да бъдат премахвани. Това в езика C++ се осъществява чрез *динамично заделяне и освобождаване на памет* за данните, с които се работи в програмата. Това се нарича *динамична памет*.

8.1. Оператори `new` и `delete`.

Динамична памет се осъществява с помощта на мениджъра на паметта. Операторът, с който се заделя памет, е `new`, а този за освобождаване на памет – `delete`. Синтаксисът на оператора `new` е следният:

```
<pointer> = new <type>[<expression>];
```

Операторът създава толкова обекти в паметта, подредени непосредствено един до друг, от тип <type>, колкото е посочено в <expression> и връща адреса на първия от тях. Изразът <expression> трябва да има стойност цяло положително число. Върнатият адрес трябва да бъде присвоен на указател от същия тип <type>, за да бъде съхранен. Операторът може да бъде извикван и така:

```
new <type>[<expression>];
```

но адресът, който връща, няма да бъде съхранен, т.е. достъп до създадения обект не може да се осъществи. Ако <expression> има стойност 1, правоъгълните скоби могат да липсват. В случай когато <expression> е със стойност по-голяма от 1, дефинираната структура представлява масив в паметта, който се нарича *динамичен*.

Синтаксисът на оператора `delete` е следният:

```
delete [<expression>] <pointer>;
```

Операторът освобождава толкова обекти в паметта, към които сочи указателят, колкото са посочени в <expression>. Ако <expression> има стойност 1, квадратните скоби могат да липсват. В случай когато последователността от обекти е едномерна, <expression> може да не се посочва – скобите се оставят празни. В този случай `delete` освобождава паметта от цялата последователност от обекти.

Ето един пример, в който се създават два обекта в паметта, използват се и след това паметта се освобождава от тях:

```
double *pi = new double;
double *r = new double;
*pi = 3.1415962;
*r = 10.0;
cout << "L: " << 2 * *pi * *r << endl;
delete pi;
delete r;
```

Трябва да се има предвид, че указателите `pi` и `r` са все още видими за функцията, в която са дефинирани. Това означава, че те могат да бъдат използвани и след като паметта, към която са сочили, е освободена. Проблемът при тяхното следващо използване е, че данните в паметта може вече да са заменени с други. Това ще доведе до логически грешки (по време на изпълнение) на програмата. Затова правилният подход е при освобождаване на паметта, към която сочи указател, той да приема стойност `NULL`. Тези два реда би трябвало да се добавят в края на предходния програмен текст:

```
pi = NULL;
r = NULL;
```

8.2. Динамични масиви.

Сега нека да видим пример, в който се създава едномерен *динамичен масив*:

```
int *DynamicArray = NULL;
unsigned N = 0;
cout << "Enter the number of elements: ";
cin >> N;
DynamicArray = new int[ N ];
if( DynamicArray == NULL ) {
      cout << "Error creating dynamic array!\n";
      exit( 0 );
}
for( int i = 0; i <= N-1; i++ ) {
      cin >> DynamicArray[ i ];
}
...processing the data of DynamicArray...
delete [] DynamicArray;
DynamicArray = NULL;
```

В този програмен текст се създава динамичен масив от цели числа, като броят на елементите се въвежда от клавиатурата. Прави се проверка за валидността на указателя `DynamicArray` (дали масивът е бил създаден). Ако масивът е бил създаден, стойностите на елементите на масива се въвеждат от клавиатурата. След това може да бъде извършена обработка на данните в масива. Накрая се освобождава паметта, заделена за масива и указателят приема стойност `NULL`.

Една друга възможност за проверка на валидност на указател е с макроса `assert`, който има следният синтаксис:

```
assert( <expression> );
```

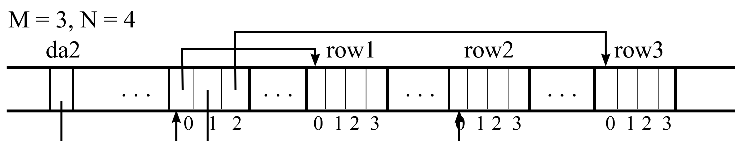
В случай, че < expression> има стойност 0 (false), се прекратява работата на програмата и в стандартния изход се извежда изразът, който е върнал стойност 0, името на файла, в който се намира макросът и номерът на реда във файла. Ако стойността на <expression> е различна от 0, програмата продължава своята работа.

Чрез указатели и динамично заделяне на памет могат да се конструират и многомерни динамични масиви. Нека да видим един

пример, който реализира динамичен двумерен масив чрез динамичен масив от указатели, като всеки от тях сочи към едномерен динамичен масив от тип int:

```
unsigned M = 0, N = 0;
cout << "Enter the number of rows: ";
cin >> M;
cout << "Enter the number of columns: ";
cin >> N;
int* *da2 = new int*[ M ];
for( int i = 0; i <= M-1; i++ )
      da2[ i ] = new int[ N ];
for( int i = 0; i <= M-1; i++ )
      for( int j = 0; j <= N-1; j++ ) {
            cout << "->";
            cin >> da2[ i ][ j ];
      }
...processing the data of da2...
for( int i = 0; i <= M-1; i++ )
      delete [] da2[ i ];
delete [] da2;
da2 = NULL;
```

Размерността на тази двумерна структура нека да бъде MxN. В програмния текст M и N се въвеждат от клавиатурата. Създава се едномерен динамичен масив (da2) от указатели към int. След което за всеки елемент на da2 се създава едномерен целочислен динамичен масив, всеки с по N елементи. Нека ги наречем row1, row2 до rowN. Така се реализира структурата, показана на фигури 8.1 и 8.2.

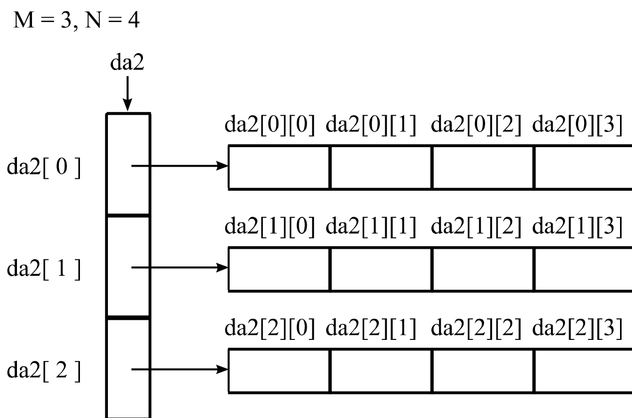


Фиг. 8.1. Реализация в паметта на двумерната динамична структура от масиви при стойности на M=3 и N=4.

Целочислените масиви са достъпни чрез da2 по следния начин: `*(da2 + 0)` ⇔ `da2[0]`, `*(da2 + 1)` ⇔ `da2[1]`, . . ,

`*(da2 + M-1)` ⇔ `da2[M-1]`. Елементите на целочислените масиви са достъпни чрез `da2` по следния начин: `*(*(da2 + i) + j)` ⇔ `da2[i][j]`, за `i=0,..,M-1`, `j=0,..,N-1`.

Освобождаването на паметта от тази структура става като първо се освободи паметта на едномерните целочислени масиви, към които сочат елементите на `da2` и след това – паметта, заета от масива `da2`.



Фиг. 8.2. Фигурно представяне на двумерната динамична структура от масиви от фигура 8.1.

Реално приложение на двумерните динамични масиви се демонстрира с реализацията на следващата задача.

Задача 8.1: Да се напише програма, която реализира произведение на две матрици. Програмата да изисква въвеждане на размерностите на двете матрици, след което проверява дали е възможно да бъде извършена операцията (броят на стълбовете на първата матрица трябва да е равен на броя на редовете на втората). Ако това е възможно, подканва потребителя да въведе данните за двете матрици, умножава ги и извежда резултата по подходящ начин.

Реализация на задача 8.1:

```
#include <iostream>
#include <iomanip>

using namespace std;
```

```
int main()
{
    //matrices:
    double **A=NULL, **B=NULL, **C=NULL;
    //A(M x N),  B(F x K):
    unsigned M = 0, N = 0, F = 0, K = 0;
    cout << "C = AB = ?\n";
    cout << "Matrix A:\n";
    cout << "Enter the number of rows    : ";
    cin >> M;
    cout << "Enter the number of columns: ";
    cin >> N;
    cout << "Matrix B:\n";
    cout << "Enter the number of rows    : ";
    cin >> F;
    cout << "Enter the number of columns: ";
    cin >> K;
    if( N != F ){
        cout << "Number of the columns of A
            does not equal to the number
            of the rows of B!!!\n";
        return 0;
    }

    cout << "Therefore, C( " << M << ", "
            << K << " ) = AB = ?\n";

    A = new double*[ M ];
    B = new double*[ N ];
    C = new double*[ M ];

    for( int i = 0; i <= M-1; i++ )
        A[ i ] = new double[ M ];
    cout << "Enter matrix A:\n";
    for( int i = 0; i <= M-1; i++ )
        for( int j = 0; j <= N-1; j++ ) {
            cout<<"A["<<i<<"]["<<j<<"]=";
            cin >> A[ i ][ j ];
        }

    for( int i = 0; i <= N-1; i++ )
        B[ i ] = new double[ K ];
```

```
    cout << "Enter matrix B:\n";
    for( int i = 0; i <= N-1; i++ )
        for( int j = 0; j <= K-1; j++ ) {
            cout<<"B["<<i<<"]["<<j<<"]=";
            cin >> B[ i ][ j ];
        }

    for( int i = 0; i <= M-1; i++ )
        C[ i ] = new double[ K ];

    //multiplication:
    double elC = 0.0;//nameren element ot C
    for( int i = 0; i <= M-1; i++ ) {
        for( int h = 0; h <= K-1; h++ ) {
            elC = 0.0;
            for( int j = 0; j <= N-1; j++ ) {
                elC += A[i][j] * B[j][h];
            }
            C[i][h] = elC;
        }
    }

    for( int i = 0; i <= M-1; i++ ) {
        for( int h = 0; h <= K-1; h++ ) {
            cout << setw(4) << C[i][h];
        }
        cout << endl;
    }

    for( int i = 0; i <= M-1; i++ )
        delete [] A[ i ];
    for( int i = 0; i <= N-1; i++ )
        delete [] B[ i ];
    for( int i = 0; i <= M-1; i++ )
        delete [] C[ i ];

    delete [] A;
    delete [] B;
    delete [] C;
        A = B = C = NULL;

    return 0;
}
```

Работата на програмата ще бъде демонстрирана с конкретни стойности на променливите за размерностите на масивите и на техните елементи:

1. Въвеждат се размерностите на матриците:
   $M = 4, N = 3, F = 3, K = 2.$
   Следователно $A_{4x3}, B_{3x2}, C_{4x2}$.
2. Въвеждат се стойности за елементите на матриците А и В:
   $$A = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 2 & 3 \\ 1 & -3 & 1 \\ 4 & 0 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 1 \\ -1 & 0 \\ 3 & 1 \end{pmatrix}.$$
3. Изчисляват се стойностите на елементите на С и се получава следният резултат:
   $$C = \begin{pmatrix} 3 & 1 \\ 7 & 3 \\ 8 & 2 \\ 14 & 6 \end{pmatrix}.$$
4. Паметта се освобождава от трите двумерни динамични масива.

Глава IX. Масиви от символи

9.1. Дефиниране на масиви от символи.

За въвеждането, съхраняването и редактирането на текстове и други символи от клавиатурата са необходими структури от данни, в които да бъдат представени. Те представляват съвкупности от символи (букви, препинателни знаци и др.), които в програмирането се наричат *символни низове* [6]. В езика C++ символните низове могат да бъдат представени по два начина. Единият е чрез *масиви от символи* (известни като C стрингове, от англ. C strings), а другият – чрез вградения в C++ клас *string*. В тази глава ще бъдат разгледани масивите от символи, а в глава 13 – символните низове от тип string.

Типът, в който се представят символи, е char. Следователно, масивите от символи трябва да бъдат дефинирани от типа char. Масивите от символи задължително трябва да съдържат *символ за край на символен низ*, чието значение ще бъде изяснено малко по-късно. Този символ е '\0'. Следователно, при дефиниране на масив от символи, освен броя на символите в масива, трябва да се предвиди и елемент за символа за край на символния низ. Ето един пример:

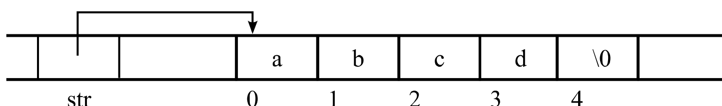
```
const int N = 4;
char str[ N+1 ];
str[ 0 ] = 'a';
str[ 1 ] = 'b';
str[ 2 ] = 'c';
str[ 3 ] = 'd';
str[ 4 ] = '\0';
```

Дефинира се масив от символи, в който се записват първите четири малки букви от английската азбука (фиг. 9.1). Тогава масивът съдържа пет елемента: четирите букви плюс символа '\0'. Затова масивът се дефинира от пет елемента (N+1, при N=4). Масивът заема пет байта в паметта, тъй като всеки символ от тип char заема един байт.

Предходният пример може да бъде реализиран и така:

```
char str[] = "abcd";
```

Това представлява инициализиране на масив от символи в момента на неговото дефиниране. В този случай броят на елементите на масива е равен на броя на символите, поставени в кавички, плюс едно. Символът '\0' се поставя като последен елемент автоматично.



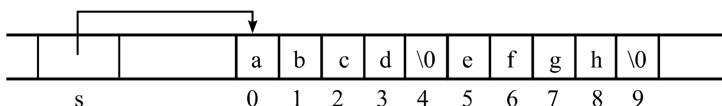
Фиг. 9.1. Разпределение на масива от символи в паметта.

Потоците в езика C++ са дефинирани да извеждат и въвеждат масиви от символи чрез директно подаване на името на масива. Те могат също да бъдат извеждани и въвеждани по елементи, но в повечето случаи това не е необходимо и в случай на въвеждане – дори по-трудно.

Извеждането на масиви от символи в командния интерпретатор с потока `cout` се прави по следния начин:

```
cout << str;
```

Това извежда целия символен низ до знака за край на символен низ, включително ако има интервали, символи за табулация, край на ред и др. Ако в масив от символи липсва знакът за край на символен низ, извеждането ще продължи извън обсега на масива, което представлява четене на данни от части на оперативната памет, които не принадлежат на масива. Затова всеки масив от символи трябва да съдържа символа '\0'. Ако символът за край е поставен преди последния елемент, стойностите на елементите след него не се извеждат, т.е. извеждането на елементи от масива се прави до първото срещане на символ за край.



Фиг. 9.2. Масив от символи в паметта.

На фигура 9.2 е показан един масив (s) от символи в паметта. Броят на елементите му е 10. Той съдържа два символа за край – единият е на позиция 4, а другият – на позиция 9. При извеждането на `s` в командния интерпретатор (`cout << s;`) ще се изведе само “abcd”.

При въвеждане на масиви от символи с потоци, въвеждането на символи се прави до появата на символ разделител (интервал, табулация) или край на ред. Това може да бъде демонстрирано със следния пример: ако е дефиниран масив от символи и потокът очаква въвеждането на име в него:

```
const N = 100;
char name[ N+1 ];
cin >> name;
```

и бъде въведено:

```
George White Blackberry
```

в `name` ще бъде записано само `"George"`.

Много важно условие при въвеждане на символи в масив е той да бъде с размерност, в която могат да бъдат записани всички символи, които му се присвояват. В противен случай опитът за записване на повече на брой символи в масива, ще доведе до промяна на стойности на байтове от паметта, които не принадлежат на масива, а това в повечето случаи ще предизвика принудително прекратяване на работата на програмата от операционната система. Това се нарича *препълване* (от англ. overflow). В предходния пример масивът `name` е дефиниран с възможност за съхраняване на 100 символа в него – място предостатъчно за символите на едно име на човек.

9.2. Вградени функции в езика за работа с масиви от символи.

Операторът за присвояване не е дефиниран за работа с масиви от символи, т.е. изрази от вида

```
str = name;
```

не копират съдържание на масив. Много често обаче в програмите се налага един масив от символи да присвоява съдържанието на друг. В този случай трябва да се извърши копиране на символите поотделно. Друг вариант е да се използва вградената в езика функция за копиране на масиви от символи с име `strcpy`.

Често броят на символите, записани в един масив, е по-малък от броя на елементите, с които той е дефиниран. При обработка на масива се работи именно с реално записаните символи в него. Тогава трябва да се знае какъв е броят им. Те могат да се преброят по следния начин:

```
int i = 0;
while( name[ i ] != '\0' )
      i++;
cout << "Number of characters: " << i << endl;
```

или чрез вградената в езика функция:

```
int i = strlen( name );
```

Тези и други функции за работа с масиви от символи са декларирани в хедъра `string.h` (`cstring`). В таблица 9.1 са дадени декларациите и описанията на някои функции от `string.h`.

Таблица 9.1. Декларации и описания на някои функции от `string.h`.

Декларация	**Описание**
char *strcpy(char *destination, const char *source);	Копира символите от source в destination включително и символа ‘\0’. *
char *strcat(char *destination, const char *source);	Добавя копие на символите на source в края на destination. *
int strcmp(const char *str1, const char *str2);	Сравнява двата масива от символи. Сравнението започва от първите символи на str1 и str2.** При първото срещане на различие в масивите от символи сравнението се прекратява и функцията връща стойност. Ако стойността е положително число означава, че str1 > str2, ако стойността е 0 – str1 = str2, а ако стойността е отрицателно число – str1< str2.
const char *strstr(const char *str1, const char *str2);	Връща адреса на първия байт от първото срещане на str2 в str1. Ако стойността на

	указателя е NULL, означава, че str2 не се среща в str1.
size_t strlen(const char *str);	Връща дължината (броя на символите) на str.

* За да се избегне препълване, размерът на destination трябва да е достатъчен, за да побере символите на source.

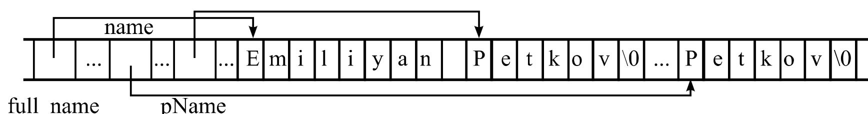
** Символите се сравняват по своите кодове в ASCII.

Задача 9.1: Да се напише програмен текст, който търси символен низ, записан в name, в масива от символи full_name. Да се извежда стойността на указателя, сочещ към намерения низ, както и символния низ, към който сочи указателят.

Реализация на задача 9.1:

```
char name[] = "Petkov";
char full_name[] = "Emiliyan Petkov";
char *pName = strstr( full_name, name );
cout << hex << (unsigned)pName;
if( pName != NULL )
       cout << " -> " << pName << endl;
else
       cout << endl << name <<" does not appear in "
              << full_name << endl;
```

Фигура 9.3 демонстрира работата на програмния текст.



Фиг. 9.3. Организация на паметта по задача 9.1.

Съществуват няколко полезни функции за работа с масиви от символи, декларирани в хедъра stdlib.h (cstdlib), които са дадени в таблица 9.2.

Таблица 9.2. Декларации и описания на някои функции от stdlib.h.

Декларация	**Описание**
char *gcvt(double value, int ndec, char *buf);	Преобразува число, подадено във value, с основа ndec, в символен низ. Записва символите в buf и връща указател към масив от символи.
double atof (const char* str);	Преобразува символен низ в число от тип double. За да е успешно преобразуването,

	символният низ трябва да започва с цифра.
int atoi (const char * str);	Преобразува символен низ в число от тип int. За да е успешно преобразуването, символният низ трябва да започва с цифра.

Следният програмен текст демонстрира употребата на поместените в таблица 9.2 функции:

```
double pi = 3.14159;
char pi_str[ 8 ];
gcvt( pi, 10, pi_str );
cout << "String: " << pi_str << endl;
double d = atof( pi_str );
cout << d << endl;
int i = atoi( pi_str );
cout << i << endl;

Изход:
String: 3.14159
3.14159
3
```

Задача 9.2. Да се напише програма, която подканва потребителя да въведе два символни низа, които се записват в програмата в `str` и `destination`. Програмата да намира и извежда колко пъти `str` се среща в `destination`.

Анализ на задача 9.2. Търсенето на `str` в `destination` може да се направи с функцията `strstr()`. Когато `str` е намерен в `destination`, тази функция връща указател, сочещ първия елемент на `str` в `destination`, т.е. указател, различен от `NULL`. Нужен ни е един указател към `char`, който първоначално да сочи в началото на `destination`. Нека той да бъде `pos`. Трябва да организираме цикъл, в който докато `pos` е различен от `NULL` търсим `str` в `pos` и ако веднага след това търсене `pos` отново е различен от `NULL`, означава, че имаме срещане. Тогава го преброяваме. В момента, в който `pos` приеме стойност `NULL`, ще означава, че `str` вече не се среща в `pos` и тогава цикълът ще прекрати своята работа.

Реализация на задача 9.2:

```
#include <iostream>
```

```
#include <string.h>
using namespace std;

int main()
{
    char str[101], destination[1001];
    cout << "Enter str:";
    cin >> str;
    cout << "Enter destination:";
    cin >> destination;
    unsigned short num = 0;
    char* pos = destination;
    while( pos = strstr( pos, str ) ) {
        num++;
        pos++;
    }
    cout << "Matchings of " << str << " in "
            << destination << ": " << num
            << " times." << endl;
    return 0;
}
```

Задача 9.3. Да се напише програма, която въвежда дума от текстовия интерфейс. Програмата да проверява дали думата е полиндром (чете се еднакво отляво надясно и отдясно наляво).

Анализ на задача 9.3. Ако думата, която е въведена е полиндром, това означава, че първият символ е еднакъв с последния, вторият – с предпоследния и така до средата на думата. Следователно, ако броят на символите в думата е `n`, то ще трябва да се направят `n/2` (целочислено деление) проверки, които можем да реализираме в цикъл. След тези проверки трябва да се излезе с решение, дали думата е полиндром или не. Това можем да кодираме в една булева променлива по следния начин: стойност `true` на променливата означава, че думата е полиндром, а стойност `false` – не е полиндром. Ако преди започването на проверката предположим, че думата е полиндром (дефинираме една булева променлива с начална стойност `true`), тогава можем да проверяваме за евентуални несъответствия в символите и ако такова бъде намерено, тогава стойността на булевата променлива да става `false` и да прекратяваме работата на цикъла с `break`. Но ако такова несъответствие не бъде регистрирано, стойността на булевата

променлива ще остане `true`. Тогава, след като цикълът завърши, може да се направи проверка на булевата променлива и ако нейната стойност е `true`, означава, че думата е полиндром, в противен случай – думата не е полиндром.

Реализация на задача 9.3:

```
#include <iostream>
#include <string.h>
using namespace std;

int main()
{
    char word[101];
    cout << "Enter a word:";
    cin >> word;
    unsigned n = strlen( word );
    bool bSame = true;
    for( int i = 0; i < n/2; i++ ){
        if( word[ i ] != word[ n-1-i ] ) {
            bSame = false;
            break;
        }
    }
    if( bSame )
        cout <<"The word is a polindrom.\n";
    else
        cout <<"The word is not a polindrom.\n";
    return 0;
}
```

Глава X. Функции

10.1. Главна функция main.

Всяка програма на C++ се състои от поне една функция и това е main. Името единствено на тази функция е фиксирано и ние не можем да го променяме, т.е. не можем да напишем главна функция с друго име (при тип конзолно приложение). Освен това тази функция се изпълнява само веднъж за едно стартиране на програмата.

Функцията main може да бъде дефинирана да връща стойност или не. Някои компилатори изискват задължително да има връщане на стойност и обикновено това е цяло число. Тази стойност се нарича *върната стойност на функцията*. Някои компилатори позволяват main да бъде дефинирана от тип void, т.е. да не връща стойност. В този случай не е нужно извикването на оператор return (той може да бъде изпълнен, но без стойност).

Функцията main може да се дефинира без параметри или с параметри. Смисълът на това главната функция да е дефинирана с параметри, е да може да приема стойности, подадени при стартирането на програмата от командния интерпретатор. Тези стойности представляват символни низове, за които не е фиксирано ограничение на броя. Затова параметрите на функцията са два: първият указва броя на символните низове (всеки въведен интервал се счита за разделител), с които се стартира програмата, а вторият съдържа символните низове (масив от указатели към масиви от символи). В този случай дефиницията на main в заглавната си част може да бъде следната:

```
int main( int argc, char** argv )
```

Параметърът argv се явява динамичен масив от указатели към char. Достъп до всеки символен низ от масива можем да получим по следния начин:

```
for( int i = 0; i <= argc-1; i++ ){
```

```
        ...processing:   argv[ i ]
    }
```

10.2. Дефиниране на функции.

Освен `main` ние можем да дефинираме и други функции в програмите си. Обикновено функциите решават по-малки задачи от поставения по-голям проблем. Също много често при писане на програмен текст се забелязват повтарящи се конструкции. Тогава е добре да се помисли за тяхното групиране във функции. Следователно, добър стил на програмиране е да умеем да предвиждаме или откриваме по-малки задачи и повтарящи се конструкции в програмите си, които да реализираме като функции. От това получаваме следните две ползи:

- дефинирайки различни функции в нашите програми, ние правим програмния текст по-ясен за разбиране, проверка и промяна;
- чрез групирането на повтарящи се фрагменти във функции се реализира оптимално използване на паметта, тъй като при стартиране на програмата, кодът, който изпълнява функцията, се записва само веднъж в паметта, след което може да се извиква многократно.

Всяка функция, различна от главната, може да бъде стартирана неограничен брой пъти, което се нарича *извикване на функцията*. Освен това вече дефинирани функции могат да бъдат групирани в отделни файлове, които да бъдат компилирани и в последствие други програми да извикват функции от тях. Така вече разработени функции, решаващи различни задачи, не е нужно да бъдат писани отново, а само извиквани и то в различни програми.

Дефинициите на функциите следват следния синтаксис:

```
type name( type par1, type par2, ...) { statements }
```

Първо трябва да бъде указан *типът на върнатата стойност*, след това се избира *име* на функцията, което трябва да бъде валиден идентификатор в C++ и последвано от заоблените скоби. Ако функцията ще приема *параметри*, тогава те се дефинират в скобите, разделени със запетаи. Накрая във фигурните скоби се поставят

операторите (statements) за изпълнение от функцията. Параметрите, с които е дефинирана функцията се наричат *формални параметри*. Блокът, определен от фигурните скоби, се нарича *тяло* на функцията.

Вече дефинирана функция може да се извика от друга функция чрез нейното име (идентификатор), последвано от заоблени скоби и ако очаква параметри, те се поставят в скобите, като се разделят със запетаи. Параметрите, които се подават на функцията при нейното извикване, се наричат *фактически параметри*. Имената на функциите представляват указатели към функциите в паметта.

Всяка функция притежава определена семантика (значение, описание). Добре е тя да бъде описана в коментар в тялото на функцията. Освен това към този коментар е добре да се добави още значение на параметрите на функцията и върнатата стойност.

Ето един пример за дефиниция на функция, която връща стойност от тип `double`, с име `Circumference`, има един формален параметър от тип `double` и е със семантика: изчислява дължината на обиколката на окръжност с радиус `R`:

```
double Circumference( double R )
{
      //Семантика:   изчислява   дължината   на
      //обиколката на окръжност с радиус R
      //Параметри: радиус R
      //Върната    стойност:    дължината    на
      //обиколката на окръжността
      const double pi = 3.1415962;
      return 2 * pi * R;
}
```

Дефинициите на функциите трябва да бъдат поставяни във файловете, съдържащи програмния текст на програмата, преди функциите, в които се извикват. Причината за това е, че компилаторът трябва да е запознат вече с обектите, които среща в програмния текст. Това обаче налага функциите, съдържащи основния код (като например `main`), да се разполагат в края на файла, което е известно неудобство.

10.3. Деклариране на функции.

Споменатият в предходната точка проблем може да се реши чрез предварително запознаване на компилатора с функциите, които се използват, като се напишат *декларации на функциите* преди тяхното извикване в други функции. В този случай декларациите се поставят в началото на файла с програмния текст, а дефинициите могат да бъдат поставени в края, като на по-предна позиция се постави главната функция. Декларацията на една функция трябва да бъде със следния синтаксис:

```
type name( type, type, ...);
```

Ето как може да се подреди програма, съдържаща функцията `Circumference`:

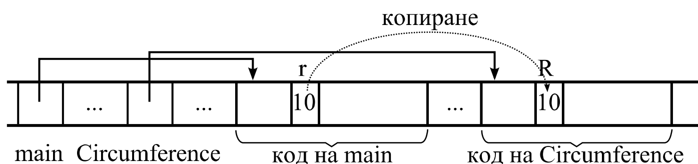
```
double Circumference( double );
. . .
int main() {
        . . .
        double r = 10.0;
        cout << Circumference( r );
        . . .
        return 0;
}

double Circumference( double R )
{
        //изчислява дължината на обиколката
        //на окръжност с радиус R
        const double pi = 3.1415962;
        return 2 * pi * R;
}
```

В програмния текст първо се помества декларацията на функцията `Circumference`, за да може тя да се използва в `main`. Дефиницията на `Circumference` е поместена след главната функция. `Circumference` се извиква в `main` с фактически параметър `10.0`. Функцията изчислява стойността `2*pi*R` и връща тази стойност в `main` на мястото, където е била извикана.

10.4. Предаване на параметри на функции по стойности.

При извикване на функция, стойностите на фактическите параметри се копират в променливите на формалните параметри на функцията. Тогава функцията се изпълнява с тези стойности. Фигура 10.1 показва едно примерно разположение на функцията `Circumference` в паметта и процесите, които се случват при нейното извикване.



Фиг. 10.1. Примерно разположение на `main` и `Circumference` в паметта.

Копирането на стойностите на фактическите параметри в променливите на формалните означава, че ако стойностите на променливите на формалните параметри бъдат променени във функцията, това няма да повлияе на стойностите на фактическите параметри. Например, ако във функцията `Circumference` стойността на `R` бъде променена, това няма да повлияе на променливата `r` от `main`. Този начин на предаване на параметри на функции се нарича *предаване по стойности*.

10.5. Предаване на параметри на функции чрез псевдоними.

Ако формалните параметри във функциите се дефинират като псевдоними, тогава нови променливи в паметта не се създават, а формалните параметри стават псевдоними на фактическите. Този начин на предаване на параметри на функции се нарича *предаване чрез псевдоними*. Така работата с формалните параметри е всъщност работа със стойностите на фактическите параметри. Това ще бъде демонстрирано чрез функция, чиято цел е да разменя стойностите на две променливи от тип `int`. Нека името на функцията е `Swap`. Ето и нейната дефиниция:

```
void Swap( int &a, int &b )
{
```

```
        int x = a;
        a = b;
        b = x;
}
```

Функцията `Swap` може да бъде извикана в `main` по следния начин:

```
int x = 5, y = 7;
Swap( x, y );
cout << x << ", " << y << endl;

Изход:
7, 5
```

Извикването на `Swap` с фактически параметри `x` и `y` предизвиква размяната на техните стойности.

10.6. Предаване на параметри на функции по адреси.

Формалните параметри на функциите могат да бъдат дефинирани като указатели. Тогава стойностите на фактическите параметри трябва да бъдат предадени с адреси. Тогава това могат да бъдат адреси на променливи или указатели. Този начин на предаване на параметри на функции се нарича *предаване по адреси*. В този случай функцията работи със стойностите на фактическите параметри във физическото място в паметта, където те се намират. Това означава, че ако във функцията се извършват действия, които променят стойностите на обектите, към които сочат формалните параметри, това всъщност предизвиква промяна на стойностите на обектите, към които сочат фактическите параметри, защото те са едни и същи обекти в паметта. За да бъде демонстрирано предаването по адреси, ще дефинираме функцията `Swap` с параметри указатели и ще видим (фиг. 10.2) разположението ѝ в паметта и процесите, които се случват при нейното извикване. Ето един вариант на нейната нова дефиниция:

```
void Swap( int *a, int *b )
{
        *a = *a ^ *b;
        *b = *a ^ *b;
        *a = *a ^ *b;
```

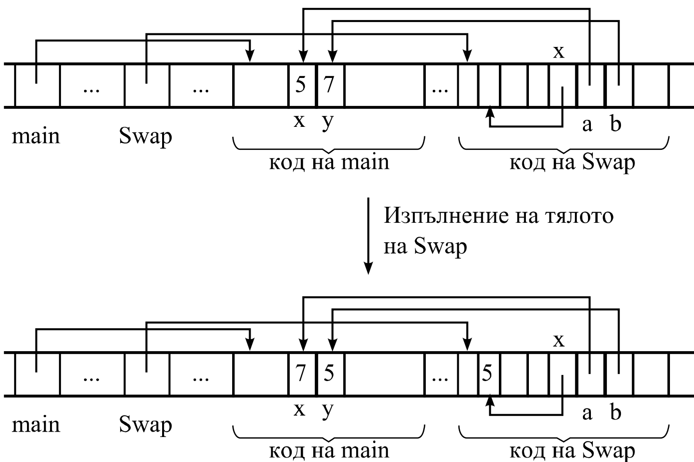
```
}
```

Функцията Swap може да бъде извикана в main по следния начин:

```
int x = 5, y = 7;
Swap( &x, &y );
cout << x << ", " << y << endl;

Изход:
7, 5
```

Функцията Swap разменя стойностите на обектите (x и y), към които сочат указателите a и b (фиг. 10.2).



Фиг. 10.2. Предаване на параметри на функции по адрес.

10.7. Предаване на масиви като параметри на функции.

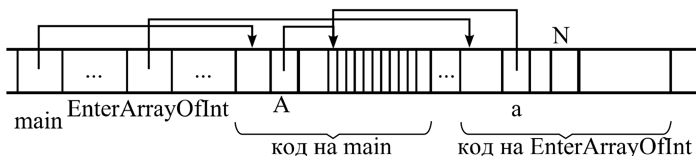
Имената на масивите представляват идентификатори на указатели в паметта. Следователно, предаването им като параметри на функции може да става само чрез предаване по адреси. В този случай, указателите, които се дефинират като формални параметри във функциите, получават адресите на фактическите параметри (указатели към масиви), което означава, че формалните параметри започват да сочат към масивите в паметта. При работа с масивите във функциите обаче, трябва да е известен броят на елементите в

масивите. Затова този брой трябва да бъде предаван чрез параметри на функциите. Ето един пример за предаване на едномерен масив от цели числа като параметър на функция:

```
void EnterArrayOfInt( int *a, const int N )
{
      for( int i = 0; i <= N-1; i++ ){
            cout << "[ " << i << " ]=";
            cin >> a[ i ];
      }
}
```

Тази функция въвежда стойности за N елемента на масив от цели числа, който е подаден на функцията (фиг. 10.3) като параметър (a). Параметърът N е дефиниран като константа, което означава, че промяна на неговата стойност не е разрешена. Функцията може да се използва по следния начин:

```
const int N = 10;
int A[ N ];
cout << "Enter array:\n";
EnterArrayOfInt( A, N );
```



Фиг. 10.3. Предаване на масиви като параметри на функции.

За да се демонстрира предаването на двумерни масиви като параметри на функции, е реализирана задача 8.1 чрез функции.

Реализация на задача 8.1 (чрез функции):

```
#include <iostream>
#include <iomanip>

using namespace std;

void NewDinamicArray2( double **, unsigned,unsigned );
```

```
void EnterAMatrix( double **, unsigned, unsigned );
void MatrixMultiplication( double **, double **,
             double **,unsigned,unsigned,unsigned );
void OutputMatrix2( double **, unsigned, unsigned );
void FreeMemoryArray2( double **, unsigned );

int main()
{
    double **A = NULL, **B = NULL, **C = NULL;
    unsigned M = 0, N = 0, F = 0, K = 0;
    cout << "C = AB = ?\n";
    cout << "Matrix A:\n";
    cout << "Enter the number of rows    : ";
    cin >> M;
    cout << "Enter the number of columns: ";
    cin >> N;
    cout << "Matrix B:\n";
    cout << "Enter the number of rows    : ";
    cin >> F;
    cout << "Enter the number of columns: ";
    cin >> K;
    if( N != F ){
        cout << "Number of the columns of A
             does not equal to the number
             of the rows of B!!!\n";
        return 0;
    }
    cout << "Therefore, C( " << M << ", " << K
             << " ) = AB = ?\n";

    A = new double*[ M ];
    B = new double*[ F ];
    C = new double*[ M ];
    NewDinamicArray2( A, M, N );
    NewDinamicArray2( B, F, K );
    NewDinamicArray2( C, M, K );

    cout << "Enter matrix A:\n";
    EnterAMatrix( A, M, N );
    cout << "Enter matrix B:\n";
    EnterAMatrix( B, F, K );

    cout << "C = A x B :\n";
    MatrixMultiplication( A, B, C, M, N, K );
    OutputMatrix2( C, M, K );

    FreeMemoryArray2( A, M );
    FreeMemoryArray2( B, N );
```

```
    FreeMemoryArray2( C, M );

    return 0;
}

void   NewDinamicArray2(   double   **X,   unsigned   M,
unsigned N )
{
    for( int i = 0; i <= M-1; i++ )
        X[ i ] = new double[ N ];
}

void EnterAMatrix( double **X,unsigned M,unsigned N )
{
    for( int i = 0; i <= M-1; i++ )
        for( int j = 0; j <= N-1; j++ ) {
            cout << "[" << i << "][" << j << "]=";
            cin >> X[ i ][ j ];
        }
}

void MatrixMultiplication( double **A, double **B,
       double **C,unsigned M,unsigned N,unsigned K )
{//C = A x B
    double elC = 0.0;
    for( int i = 0; i <= M-1; i++ ) {
        for( int h = 0; h <= K-1; h++ ) {
            elC = 0.0;
            for( int j = 0; j <= N-1; j++ ) {
                elC += A[i][j] * B[j][h];
            }
            C[i][h] = elC;
        }
    }
}

void OutputMatrix2( double **X,unsigned M,unsigned N )
{
    for( int i = 0; i <= M-1; i++ ) {
        for( int h = 0; h <= N-1; h++ ) {
            cout << setw(4) << X[i][h];
        }
        cout << endl;
    }
}

void FreeMemoryArray2( double **X, unsigned N )
{
```

```
    for( int i = 0; i <= N-1; i++ )
        delete [] X[ i ];
    delete [] X;
    X = NULL;
}
```

Сега ще разгледаме случая, в който масиви от символи се подават като параметри на функции. Характерно за масивите от символи е, че те притежават символ за край на символния низ в тях. Следователно, параметър, указващ броя на елементите в тях, няма нужда да се подава. Ето един пример, демонстриращ това:

```
char ch[ 101 ] = "akjshgdjhas";
. . .
void StrEdit( char *c )
{
      unsigned n = strlen( c );
      for( unsigned i = 0; i <= n-1; i++ ){
            . . . processing c[ i ] . . .
      }
}

. . .
StrEdit( ch );
```

Понякога се налага (особено при обработка на масиви от символи) функции да връщат указатели. Това трябва да бъде указано още при тяхното дефиниране.

Задача 10.1. Да се напише функция, която приема като параметри масив от символи и един символ за търсене. Функцията да връща указател, който сочи към елемента на първото срещане на този символ в масива, а ако символът не се среща – указателят да има стойност NULL.

Реализация на задача 10.1:

```
char *SearchCharacter( char *s, char c )
{
    unsigned i = 0;
    while( s[i] != '\0' ) {
        if( s[i] == c )
            return &s[i];
        i++;
```

```
    }
    return NULL;
}
```

Функцията `SearchCharacter` може да бъде извикана в друга функция, като адресът, който тя връща, да бъде присвоен на указател от съответния тип по следния начин:

```
char *pCh = SearchCharacter( "abcdefg", 'c' );
cout << pCh << endl;
```

10.8. Област на действие на променливи и константи.

В езика C++ променливи и константи могат да бъдат дефинирани почти на всяко място в програмата. Това, което трябва да се знае, е как се определя областта на действие на всяка една от тях.

Областта на действие на всяка дефинирана променлива или константа започва непосредствено след нейната дефиниция, т.е. променливи и константи не могат да бъдат използвани в програмния текст преди техните дефиниции.

В програмите могат да бъдат дефинирани глобални променливи и константи, което обикновено се прави в началото на програмния текст непосредствено под частта на заглавните файлове. В този случай те могат да се използват (имат област на действие, „видими" са) в целия програмен текст по-надолу.

Формалните параметри на функциите са с област на действие цялата функция, в която са дефинирани. Обектите на формалните параметри се създават в паметта веднага след извикването на функцията и се премахват оттам непосредствено преди завършването ѝ.

Дефинираните променливи и константи в дадена функция се наричат *локални*. Когато те са дефинирани в тялото на функция (извън тела на оператори), те са видими до края на функцията за всички оператори в нея. Ако те са дефинирани в тялото на оператор, те са видими от мястото на своето дефиниране до края на тази част. Например:

```
if( a > 0 ) {
    int j = 0;
```

```
        . . .
        j++;
}
else {
        . . .
        cout << j;   ← Грешка!
}
```

Ако променлива или константа е дефинирана в частта за изрази в даден оператор, тогава тя е видима в неговото тяло. Някои компилатори (не GCC) я правят видима и след тялото на оператора, но само в частта, в която той е вложен. Например:

```
for( unsigned i = 0; i <= n; i++ ) {
        cout << i;
}
i--;  ← Тук е видима само при някои компилатори!
```

Глава XI. Указатели към функции. Разпределение на паметта. Вградени и предефинирани функции. Рекурсия

11.1. Указатели към функции.

За да може една функция да бъде изпълнена от централния процесор, нейният код се разполага на определено място в паметта. Следователно, всяка функция има начален адрес в паметта, откъдето започва нейният код. В C++ можем да работим директно с адресите на функциите в паметта, като ги присвояваме на указатели, които в този случай се наричат *указатели към функции*.

Нека е дефинирана функцията `Sum`, която намира сумата $S = x + \frac{x}{2} + \frac{x}{3} + \cdots + \frac{x}{n}$, където `x` е реално число, а `n` е цяло положително число, като и двете се подават като параметри на функцията:

```
double Sum( double x, unsigned n )
{
    double S = 0.0;
    unsigned i = 1;
    while( i <= n ){
        S += x / i;
        i++;
    }
    return S;
}
```

В друга функция може да бъде дефиниран указател към функцията `Sum` и тя да бъде извикана чрез него по следния начин:

```
double (*pFunc)( double, unsigned );
pFunc = Sum;
cout << pFunc( 2.5, 10 ) << endl;
```

Адресът на функцията Sum се присвоява на указателя pFunc по този начин, тъй като името на всяка функция представлява указател към кода ѝ в паметта.

Указателите към функции се използват предимно когато функции трябва да бъдат предавани като параметри на други функции. Това се налага, когато дадена функция, наричана *регистрираща* (от англ. registering), трябва да вика друга функция, наричана *изпълнима* (на англ. callback), и в момента на компилиране на програмата втората не е известна, а само какви параметри ще има и от какъв тип е върнатата ѝ стойност. Смисълът на тези термини е, че регистриращата функция регистрира едно изпълнение на програмен код, което ще се случи след като бъде поискано. Това може да се осъществи само чрез указател към функция (изпълнима), като стойността на указателя се предава като параметър на регистриращата функция.

Можем да декларираме регистрираща функция (Reg) като евентуално изпълнимата функция да бъде Sum:

```
void Reg( double (*)( double, unsigned ) );
```

А ето как би могла да изглежда дефиницията ѝ:

```
void Reg( double (*p)( double, unsigned ) )
{
    cout << p( 2.5, 10 ) << endl;
}
```

В този случай извикването на Reg може да стане така:

```
Reg( Sum );
```

11.2. Разпределение на паметта.

Всяка програма преди да бъде изпълнена от централния процесор се помества в оперативната памет и по време на своето изпълнение тя се нуждае от допълнително пространство, в което да записва данните, с които работи. Разпределението на паметта за една програма се разделя на четири области: *област за програмния код*, *област за глобалните променливи*, *стек* (от англ. stack) и *хийп* (от

англ. heap). В областта за програмния код се помества компилираната програма, т.е. машинния код, който процесорът ще изпълнява. Кодът на всяка функция се намира на определен адрес в паметта.

В областта за глобалните променливи се поместват обектите на глобалните променливи, дефинирани в програмата. Стекът е мястото в паметта, където се разполагат адресите на функциите, параметрите и локалните променливи. Хийпът представлява голямо пространство от паметта, където се разполагат обектите, създадени динамично.

Организацията на стека е винаги следната: последният обект, който е влязъл, първи излиза. Когато една функция е извикана за изпълнение, адресът ѝ се копира в стека. След това се разполагат обектите на параметрите на функцията и локалните променливи в нея. При завършване на функцията обектите последователно се премахват от стека, като се започва от последния и се стигне до първия, който всъщност е адресът на функцията в паметта. Това ще бъде демонстрирано с конкретен пример. Нека да имаме следната програма:

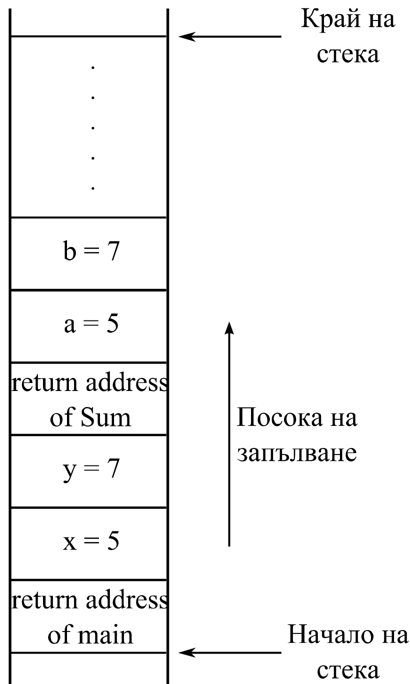
```
#include <iostream>
using namespace std;

void Sum( int a, int b )
{
	return a + b;
}

int main()
{
	int x = 5, y = 7;
	cout << Sum( x, y ) << endl;
	return 0;
}
```

При нейното стартиране кодът на `main` и кодът на `Sum` се зареждат в паметта. В стека последователно се създават обекти за: адреса на `main`, стойността на `x`, стойността на `y`, адреса на `Swap`, стойността на `a`, стойността на `b` (фиг. 11.1). След като се изчисли сумата на `a` и `b`, стекът се освобождава от тези променливи, стойността, изчислена

от `Sum`, се връща в `main`, след това последователно стекът се освобождава от адреса на `Sum`, от стойностите на `x` и `y` и накрая и от адреса на `main`.



Фиг. 11.1. Запълване на стек.

Обемът на стека зависи от конкретния компилатор и системата, на която се компилира програмата, но той винаги е сравнително малък. Дори за не толкова голям обем данни стекът може да бъде препълнен. Това води до грешката *препълнен стек* (от англ. stack overflow). Например, ако в една функция дефинираме масива

```
long double ld[ 1000000 ];
```

за повечето системи и компилатори това ще породи грешката stack overflow при изпълнението на програмата.

При използване на оператора `new` се създават обекти в хийпа. В хийпа обектите не се разполагат последователно. Новите обекти се разполагат там, където има място за тях. Тъй като първоначално не е ясно, къде ще бъде разположен новият обект, операторът `new` връща указател. Когато операторът `delete` бъде използван за даден указател, паметта, към която той е сочел, се връща на хийпа.

11.3. Вградени функции.

Функция може да бъде дефинирана така, че кодът ѝ да се копира на всяко място в паметта, където тя се извиква. Това е направено с цел да се предотврати копирането на адреса на функцията в стека, предаването на параметрите и извършването на обръщението към местоположението на функцията. Такава функция се нарича *вградена* (от англ. inline function). За да укажем, че една функция ще бъде вградена, дефиницията трябва да започва с ключовата дума `inline`. Например:

```
inline void PrintEndOfLine() { cout << '\n'; }
```

Използването на вградени функции е оправдано, когато се знае, че кодът им ще бъде много кратък. В противен случай употребата им ще доведе до прекалено дублиране на един и същи код в стека.

11.4. Предефинирани функции.

При реализирането на еднакви по смисъл операции в една програма, могат да бъдат дефинирани функции с едни и същи имена, но с различен брой и/или различен тип параметри. Такива функции се наричат предефинирани (от англ. overloading functions). Следният програмен текст илюстрира дефинирането на две предефинирани функции:

```
int Sum( int a, int b ) { return a+b; }
double Sum( double a, double b ) { return a+b; }
```

11.5. Рекурсия.

Когато една функция извиква сама себе си, то тя се нарича *рекурсивна*, а механизмът за това – *рекурсия*. Рекурсията може да

бъде *директна* (пряка) – в случай, когато функцията извиква себе си директно в своето тяло и *индиректна* (непряка) – когато функцията извиква друга функция, която извиква първата. Рекурсията се прилага най-вече за реализирането на алгоритми, които са рекурсивни по дефиниция. Един от най-добрите примери за рекурсия е реализацията на функция, която намира факториел на число. Дефиницията на факториел е рекурсивна:

$$n! = \begin{cases} 1, & n = 0 \text{ или } 1 \\ n.(n-1)!, & n > 1 \end{cases}$$

Следователно, реализацията може да бъде рекурсивна. Ето как може да изглежда функцията, която намира факториел на число, подадено като неин параметър:

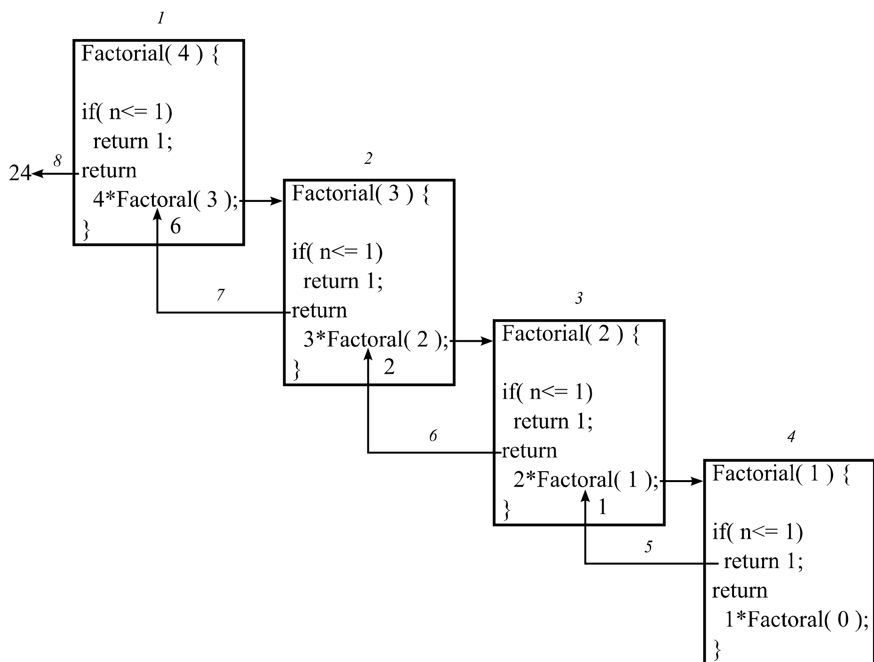
```
unsigned Factorial( unsigned n )
{
    if( n <= 1 )
        return 1;
    return n * Factorial( n-1 );
}
```

Функцията може да бъде извиквана по следния начин:

```
cout << Factorial( 4 ) << endl;

Изход:
24
```

Използваният тип на данните във функцията е `unsigned`, следователно стойността на фактическия параметър не трябва да бъде по-голяма от 13. В противен случай факториелът няма да се изчислява правилно, тъй като максималната стойност на цяло число в този тип е $2^{32}-1$, а 13! = 1932053504.



Фиг. 11.2. Извикване на функцията `Factorial` със стойност на фактическия параметър 4.

Нека да видим какво се случва при извикване на функцията `Factorial` със стойност на фактическия параметър 4 (фиг. 11.2):

1. Стойността на условието в условния оператор е `false`. Изпълнява се return `n * Factorial( n-1 );`, следователно отново се извиква функцията `Factorial` със стойност на фактическия параметър 3.
2. Стойността на условието в условния оператор е `false`. Изпълнява се return `n * Factorial( n-1 );`, следователно отново се извиква функцията `Factorial` със стойност на фактическия параметър 2.
3. Стойността на условието в условния оператор е `false`. Изпълнява се return `n * Factorial( n-1 );`, следователно отново се извиква функцията `Factorial` със стойност на фактическия параметър 1.

4. Стойността на условието в условния оператор е `true`. Изпълнява се `return 1;`, следователно това копие на функцията връща стойност 1 и се премахва от стека.
5. Връща се стойност 1 в извикващата функция. Тя изчислява 2 * 1, връща стойност 2 и се премахва от стека.
6. Връща се стойност 2 в извикващата функция. Тя изчислява 3 * 2, връща стойност 6 и се премахва от стека.
7. Връща се стойност 6 в извикващата функция. Тя изчислява 4 * 6, връща стойност 24 и се премахва от стека.

Доказано е, че всеки рекурсивен алгоритъм може да бъде реализиран и нерекурсивно (например чрез цикъл). Рекурсията винаги изисква повече памет и изпълнението ѝ е по-бавно от нерекурсивния вариант на алгоритъма. Затова употребата ѝ трябва да бъде внимателно обмислена. Ето как може да бъде реализирана функцията за намиране на факториел без употребата на рекурсия:

```
unsigned Factorial( unsigned n )
{
    if( n <= 1 ) return 1;
    unsigned f = 1, i = 1;
    while( i <= n ) {
        f *= i;
        i++;
    }
    return f;
}
```

В този случай програмният текст на функцията е малко повече, но използваната памет ще бъде по-малко и изпълнението – по-бързо.

Глава XII. Структури и класове

Съставен тип данни е този, който се състои от множество компоненти от различни типове данни, т.е. в неговите компоненти могат да се съхраняват данни от различни (вече дефинирани) типове (стандартни и съставни) [6].

12.1. Структури.

В езика C++ могат да бъдат дефинирани *структури* от данни, представляващи групи от компоненти, дефинирани от стандартни типове данни и/или вече дефинирани структури. Следователно, структурите служат за дефинирането на съставни типове данни. Дефинираните компоненти в структура могат да бъдат променливи и се наричат *полета* или още *член-променливи* (от англ. member variables). Така дефинираната структура представлява нов тип данни в програмата, от който могат да се дефинират *екземпляри* (променливи), наречени още *обекти*. Дефинирането на структура се прави чрез ключовата дума `struct`, следвана от идентификатор (име на структурата) и фигурни скоби, в които се дефинират полетата:

```
struct <identifier> {
      <type> <identifier_1>;
      . . .
      <type> <identifier_n>;
};
```

Освен полета в структурите могат да бъдат дефинирани функции, наречени *член-функции* или още *методи*. Те са предвидени да работят с полетата на структурата. В този случай синтаксисът е следният:

```
struct <identifier> {
      дефиниране на полета
      дефиниране на методи
};
```

Полетата и методите на дадена структура (по подразбиране) са достъпни за външни функции чрез дефинирания обект. Това може да бъде контролирано чрез използване на режимите за достъп: `private`, `protected` и `public`.

Един метод в структурата е по-специален от останалите и се нарича *конструктор*. Конструкторът не връща стойност (тип на върнатата стойност не се указва), може да бъде дефиниран с параметри или без и името му трябва да съвпада с името на структурата. В една структура могат да бъдат дефинирани няколко конструктора, но всички трябва да носят името на структурата, като се различават по типовете и/или броя на параметрите, т.е. да са предефинирани. Конструкторът без параметри се нарича *конструктор по подразбиране*. При дефиниране на обект от структура винаги се извиква един от конструкторите. Кой от тях ще се извика се определя от начина на дефиниране на новия обект. Конструкторите не могат да бъдат викани явно.

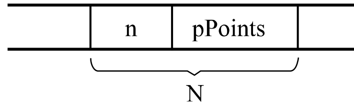
Много често се налага да разработваме програми, които работят с по-сложни обекти. Например, една чертожна програма трябва да чертае правилни многоъгълници. Това са обекти, които трябва да бъдат представени по подходящ начин в програмата. Но тези многоъгълници се състоят от отсечки, а отсечките се чертаят от една точка до друга. Следователно, в програмата е добре да съставим структури, описващи точки и многоъгълници. Ето какви структури можем да дефинираме в този случай:

```
struct Point {
    float x, y;
};
struct Ngon {
    unsigned n;//number of points
    Point *pPoints;//dynamic array of Point
    Ngon() { n = 0; pPoints = NULL; };
};
```

Структурата `Ngon` се състои от две полета (фиг. 12.1), като едното от тях е също с две полета, защото използваме `Point` за дефинирането на `Ngon`. Дефинирането на обект от `Ngon` (например в `main`) можем да направим така:

```
Ngon N;
```

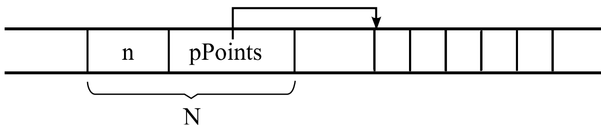
Създава се обекта N (многоъгълник) и се извиква конструкторът Ngon() на структурата, който задава начални стойности на полетата.



Фиг. 12.1. Полетата на N.

Достъп до полетата получаваме като използваме точка:

```
N.n = 5;
N.pPoints = new Point[ N.n ];
за i = 0, .. ,N.n-1 задаване на стойности на:
      N.pPoints[ i ].x
      N.pPoints[ i ].y
```



Фиг. 12.2. Разпределение на паметта за полетата на N
и данните към които сочат.

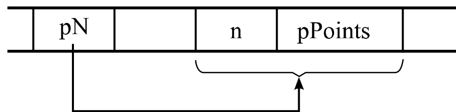
Тъй като тук се използва операторът new, редно е да се запитаме кога заделената памет трябва да бъде освободена. Това можем да направим по всяко време, но има възможност да бъде направено и автоматично, непосредствено преди унищожаване на обекта (N). За целта трябва да бъде дефиниран специален метод, наречен *деструктор*. В една структура може да има само един деструктор. Дефинирането му започва със символа ~, последван от име, което задължително е името на структурата. Деструкторите не могат да притежават параметри. Извикват се автоматично при унищожаване на обекти на структурата.

Ето как ще изглежда структурата с деструктор:

```
struct Ngon {
      unsigned n;//number of points
      Point *pPoints;//dynamic array of Point
      Ngon() { n = 0; pPoints = NULL; };
      ~Ngon() {
            if( pPoints != NULL ){
                  delete [] pPoints;
                  pPoints = NULL; } };
};
```

Обект на дадена структура може да бъде създаден и чрез указател (фиг. 12.3) по следния начин:

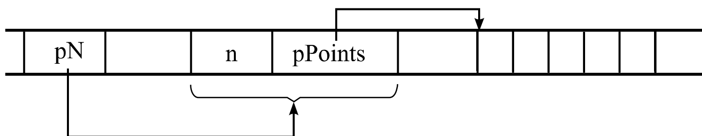
```
Ngon *pN = new Ngon;
```



Фиг. 12.3. Дефиниране на указател, сочещ към обект от структурата.

При създаването на обекта се извиква конструкторът на структурата, а при неговото унищожаване – деструкторът. В този случай достъпът до полета (фиг. 12.4) става по следния начин:

```
pN->n = 5;
pN->pPoints = new Point[ pN->n ];
за i = 0, .. ,pN->n-1 задаване на стойности на:
      pN->pPoints[ i ].x
      pN->pPoints[ i ].y
. . . работа с компонентите на pN . . .
delete pN;//Освобождаване на динамичната памет
```



Фиг. 12.4. Разпределение на паметта за полетата на обекта, към който сочи `pN`.

В структурата `Ngon` можем да ограничим достъпа до полетата като ги дефинираме `private`. В този случай трябва да дефинираме конструктора с параметри и създаването на динамичния масив да става в него, както и изчисляването на точките (върховете) на правилния многоъгълник. Последното може да стане по зададен радиус. Ето как ще изглежда структурата в този случай:

```
struct Ngon {
private:
      unsigned n;
      float R;//radius
      Point *pPoints;
public:
      Ngon( unsigned num, float r ) {
            n = num;
            R = r;
            pPoints = new Point[ n ];
            double t = 0.0, pi = 3.14159265;
            double step = 2*pi / n;
            for( int i=0; i<n; i++ ) {
                  pPoints[ i ].x = R*cos( t );
                  pPoints[ i ].y = R*sin( t );
                  t += step;
            }
      };
      ~Ngon() {
            if( pPoints != NULL ) {
                  delete [] pPoints;
                  pPoints = NULL; }
      };
};
```

При така дефинираната структура, обектът ще се създава готов за подаване на функция за чертане, тъй като той ще съдържа координатите на всичките си върхове:

```
Ngon *pN = new Ngon( 6, 10.0 );
. . .пдаване на pN на функция за чертане. . .
delete pN;
pN = NULL;
```

При освобождаване на паметта, към която сочи `pN`, автоматично ще се извика деструкторът на структурата, при което паметта ще бъде освободена и от динамичния масив. При дефиниране на полетата с режим на достъп `private`, обръщения от вида

```
pN->n = 6;
```

са недопустими.

Работата със структури в програмирането има две основни предимства:

- данните могат да бъдат структурирани в единни структури, от които могат да бъдат създавани множество обекти;
- дефинират се специфични методи, с които се обработват данните на обектите.

В разработването на програми рядко се използват всички възможности, които предоставят структурите. Обикновено те служат за дефинирането на съставни типове данни, състоящи се само от полета, дефинирани от стандартни и/или вече дефинирани съставни типове. Възможността в структурите да се дефинират методи се използва рядко. За тази цел се ползват класовете.

12.2. Класове.

Класовете в C++ представляват съставни типове данни, които могат да бъдат дефинирани от програмистите. Чрез тях се описват обекти от реалния свят или абстрактни. От тази възможност произлиза и терминът *обектноориентирано програмиране*. Класовете се състоят от множество компоненти, които могат да бъдат данни и функции. Данните, които се дефинират в клас се наричат *член-променливи*, а функциите *член-функции* или *методи*. Дефинирането на клас се прави чрез ключовата дума `class`, следвана от идентификатор (име на класа) и фигурни скоби, в които се дефинират компонентите. Синтаксисът е следният:

```
class <identifier> {
      дефиниране на полета
      дефиниране на методи
};
```

Един дефиниран клас се предполага да съдържа определени методи за обработка на данните и представлява нов съставен тип в програмата, от който могат да се дефинират екземпляри, наречени *обекти*.

Всеки клас притежава поне един *конструктор* (макар и да не е зададен явно), който се извиква при създаването на обект. Конструкторът не връща стойност (тип на върната стойност не се указва), името му трябва да съвпада с името на класа, може да бъде дефиниран с параметри или без. В един клас могат да бъдат дефинирани няколко конструктора, но всички трябва да носят името на класа, като се различават по типовете и/или броя на параметрите. Конструкторът без параметри се нарича *конструктор по подразбиране*.

При дефиниране на обект от даден клас винаги се изпълнява един от конструкторите. Кой от тях ще се извика се определя от начина на дефиниране на новия обект. Класът може да има *деструктор*. Употребата му се налага, ако има динамична памет, която трябва да бъде освободена непосредствено преди унищожаването на обекта. В един клас може да има само един деструктор. Дефинирането му започва със символа ~, последван от име, което задължително е името на класа. Деструкторите не могат да притежават параметри. Извикват се автоматично при унищожаване на обекти на класовете.

Обект на даден клас може да бъде създаден и чрез указател.

Методите на класа имат пряк достъп до член-променливите му. Достъпът до компонентите на класа във външни (за класа) функции се регламентира при самото дефиниране на класа. Режимите на достъп могат да бъдат: `private`, `protected` и `public`. Ако един компонент е дефиниран като `private`, той ще бъде видим само за методите на класа, а ако е дефиниран като `protected` или `public`, той ще бъде достъпен и във външни функции. Разликата в режимите `protected` и `public` се явява във възможността класове да бъдат наследявани от други класове, а с тях и компонентите им.

При дефинирането на клас се декларират неговите методи. Декларациите им се наричат още *прототипи*. Дефинициите на методите обикновено се поместват след дефиницията на класа. За да се укаже принадлежността на метод към даден клас се използва

знакът за принадлежност „`::`”. Методите също могат да бъдат дефинирани в дефиницията на класа. Тогава те се наричат *вмъкнати методи* (от англ. inline member-functions). Вмъкнатите методи обикновено са функции с много кратък програмен текст.

Една от основните идеи в обектноориентираното програмиране е капсулирането на данните, т.е. данните, които обектът притежава, да бъдат скрити от този, който ползва класа. Предполага се тези данни да бъдат обработвани само или предимно чрез методите на класа. Затова при дефиниране на компонентите в класовете те по подразбиране са `private`. Това е съществената разлика между класовете и структурите, в които дефинираните компоненти по подразбиране са `public`.

Задача 12.1. Да се дефинира клас с име `Student`, чиито компоненти описват един студент с име, факултетен номер (цяло число) и среден успех. Класът да притежава конструктор по подразбиране, методи за присвояване стойности на член-променливите, извличане на стойностите на член-променливите, извеждане на данните за обекта в конзолата, търсене по име и търсене по факултетен номер. Да се напише функция `main`, която демонстрира употребата на `Student`.

Реализация на задача 12.1:

```
#include <iostream>
using namespace std;

class Student {
    string name;
    unsigned fnum;
    double score;
public:
    Student(){ name=""; fnum=0; score=0.0;};
    void SetData(const string,unsigned, double);
    string GetName(){ return name; };
    unsigned GetFNum() { return fnum; };
    double GetScore() { return score; };
    void ConsoleOutput();
    bool NameIsFound( string );
    bool FNumIsFound( unsigned );
};
```

```
void Student::SetData( const string n,
                        unsigned f, double s )
{
    name = n; fnum = f; score = s;
}
void Student::ConsoleOutput()
{
    cout << "Name:       " << name << endl;
    cout << "Faculty ID: " << fnum << endl;
    cout << "Score:      " << score << endl;
}
bool Student::NameIsFound( string n )
{
    if( name.find( n ) != string::npos )
            return true;
    return false;
}
bool Student::FNumIsFound( unsigned n )
{
    if( n == fnum ) return true;
    return false;
}
int main()
{
    Student S;
    S.ConsoleOutput();
    S.SetData( "Ivan Petrov", 13759, 5.67 );
    S.ConsoleOutput();
    cout << "\nFuculty ID only: "
              << S.GetFNum() << endl;

    string n = "Petrov";
    cout << "\nAppearance of " << n << ":\n";
    if( S.NameIsFound( n ) )
        cout << "Yes!\n";
    else
        cout << "No!\n";

    cout << S.FNumIsFound( 12000 ) << endl;

    return 0;
}
```

Глава XIII. Клас `vector`. Клас `string`

В С++ са реализирани специални носители, които съхраняват колекция от обекти (техните елементи), позволявайки голяма гъвкавост в поддържаните типове [5]. Наричат се *контейнери* (на англ. containers).

13.1. Клас `vector`.

Класът `vector` представлява контейнер и служи за представяне и съхраняване на съвкупности от данни. Всеки обект от този клас можем да наричаме *вектор*. Всеки вектор представлява съвкупност от еднотипни данни. Елементите на вектора се разполагат последователно един след друг в паметта.

Като представители на структури за представяне на съвкупности от еднотипни данни дотук разгледахме масивите. Те имат един съществен недостатък, а именно невъзможността динамично да се променя размерността им. Много задачи обаче налагат динамичното увеличаване и намаляване на съвкупността от елементи. Това се налага в два случая:

- когато първоначално броят на постъпващите елементи в паметта не е известен и той трябва да се определи по време на изпълнение на програмата;
- когато след като е създадена съвкупността от елементи се налага добавяне на елемент в нея или премахване на такъв.

Класът `vector` позволява динамично разширяване и намаляване на броя на елементите в своите обекти. По този начин може да се реализира оптимално използване на паметта. Класът постига това чрез използването на динамични масиви и методи за преразпределение на паметта.

13.1.1. Дефиниране на вектори.

Синтаксисът за дефиниране на празен вектор е следният:

```
vector<type> identifier;
```

а за вектор със `size` на брой елементи е следният:

```
vector<type> identifier( size );
```

Нека дефинираме един вектор, в който един студент да може да съхрани своите оценки от взетите си пет изпита:

```
vector<double>  vEvaluation( 5 );
```

Дефиниран е вектор с пет елемента. Всеки елемент на вектора може да приеме стойност по следния начин:

```
for( i = 0; i <= 4; i++ )
      cin << vEvaluation[ i ];
```

Индексирането на елементите на вектора винаги започва от 0 и се прави в правоъгълни скоби (както при масивите).

Нека сега дефинираме един вектор, в който всеки студент да може да съхрани своите оценки от изпитите си до момента:

```
vector<double>  vEvaluation;
```

Дефиниран е празен вектор. Тъй като всеки студент може да има различен брой взети изпити, броят на елементите на вектора при дефинирането му не може да се определи. В този случай въвеждането на елементи във вектора може да се направи с добавяне. Методът за това е `push_back()`. Той променя размера на вектора като добавя елемент в края му. Ето как можем да добавим елемент към `vEvaluation`:

```
double d;
cin >> d;
vEvaluation.push_back( d );
```

Ако този код бъде поставен в циклична структура, могат да бъдат добавени във вектора толкова елементи, колкото е необходимо.

Нека сега да разгледаме следния случай: знаем, че оценките се обработват от последната към първата и искаме всяка обработена оценка да освободи вектора веднага. Това можем да направим с

метода `pop_back()`, тъй като той премахва последния елемент от вектора.

```
vEvaluation.pop_back();
```

Този метод не връща стойност. Ако стойността на последния елемент е необходима, тя трябва да бъде извлечена преди премахването му. Например така:

```
double last = vEvaluation[vEvaluation.size()-1];
```

Методът `size()` връща броя на елементите във вектора. Този брой се нарича *размер на вектора.*

13.1.2. Итератори.

Итератор е всеки обект, който сочейки някакъв елемент в редица от елементи (масив или контейнер), има способността да премине през тях в определен диапазон с помощта на набор от оператори (например с инкрементиране (++) и декрементиране (--)).

Най-ясен пример за итератор е указател, който сочи към елемент на масив. Той може да премине през неговите елементи чрез операторите за инкрементиране и декрементиране, като се прилага адресна аритметика.

Всеки контейнер има специфичен тип итератор, който е проектиран да преминава през неговите елементи. Тук не можем да използваме адресната аритметика от вида:

```
*(vEvaluation + i)  ←  Грешка!
```

За да се преминава през елементите на вектор трябва да се дефинира итератор. Това се прави със следния синтаксис:

```
vector<type>::iterator <identifier>;
```

За итераторите може да бъде прилагана адресна аритметика.

Ето един пример на програма, в която се демонстрира употребата на итератор:

```
#include <iostream>
#include <vector>
```

```
#include <cstdlib>
using namespace std;

int main()
{
  vector<int> vec;
  for (int i=1; i<=5; i++)
      vec.push_back( rand() );
  vector<int>::iterator it;
  cout << "The vector contains:\n";
  for( it = vec.begin(); it != vec.end(); it++ )
    cout << *it << endl;

  return 0;
}
```

В този пример се дефинира празен вектор за цели числа, който впоследствие приема пет цели псевдослучайни числа. Дефинира се итератор (`it`). Използват се методите `begin()` и `end()`, за да се върнат итератори съответно към първия и последния елемент на вектора. В цикъл се обхождат всички елементи на вектора (от първия към последния), като се извежда стойността на всеки.

13.1.3. Основни методи на класа `vector`.

Класът `vector` съдържа множество методи за работа с вектори. Част от тях вече бяха споменати, а други са поместени в таблица 13.1.

Таблица 13.1. Прототипи и описания на някои методи от класа `vector`.

Прототип	**Описание**
size_type size() const;	Връща броя на елементите във вектор.
size_type max_size() const;	Връща максималния брой елементи, които могат да бъдат съхранени във вектора. Нарича се капацитет.
size_type capacity() const;	Връща броя на елементите, които могат да бъдат записани във вектора, без да се налага разширяване на капацитета му.
void resize (size_type n, value_type val = value_type());	Увеличава капацитета на вектора с n на брой елемента.
bool empty() const;	Връща `true`, ако векторът е празен, и `false` в противен случай.
void reserve(size_type n);	Заявява капацитета на вектора да бъде

	достатъчен за съхраняването на поне n на брой елемента.
iterator insert(iterator pos, const value_type& val);	Вмъква нов елемент във вектора на позиция pos.*
iterator erase(iterator pos);	Премахва елемент от вектора на позиция pos.*
void swap(vector& x);	Разменя съдържанието на вектора с това на x, който е друг вектор с елементи от същия тип, като капацитетите им могат да се различават.

* Предефиниран метод.

Типът `size_type`, който е използван в класа `vector`, не може да бъде използван за дефиниране на променливи в програмите, а е еквивалентен на `unsigned int`.

13.2. Клас `string`.

Освен в масиви от символи, символни низове могат да бъдат представяни и в обекти от вградения в C++ клас `string`. Всеки обект от този клас представлява вектор от символи. Всеки обект, дефиниран от `string`, може да присвоява стойност символен низ или стойност на друг обект от `string`. Обекти на класа `string` можем да наричаме *стрингове*.

13.2.1. Дефиниране на стрингове.

Реализацията на стринговете е направена чрез вектори. Затова при работа със стрингове не е нужно да се определя явно размерността им. За динамичното увеличаване на капацитета се грижи метод от класа.

При записването на символен низ в стринг, след последния записан символ, автоматично се поставя символът за карай '\0'. Стрингове могат да бъдат присвоявани на други стрингове чрез оператора за присвояване. Стрингове могат да бъдат свързвани в един чрез оператора '+', като резултатът се присвоява също на стринг. Към стринг може да бъде добавен стринг или символен низ чрез оператора '+='. При извеждане на стрингове с потоци се извеждат всички символи от стринга до символа '\0'. Ето един пример за дефиниране, присвояване, добавяне и извеждане на стрингове:

```
string s1 = "abc";
string s2 = "def";
```

```
string s = s1 + s2;
s += "0123456789";
cout << s;

Изход:
abcdef0123456789
```

При въвеждане на стрингове с потока `cin` се въвеждат символите до срещането на символите за интервал, табулация или край на ред.

Лесно могат да бъдат създавани масиви от стрингове. Ето един пример:

```
string a_str[3] = { "abc", "defg", "hijkl" };
for( int i = 0; i <= 2; i++ )
      cout << a_str[ i ];
```

13.2.2. Основни методи на класа `string`.

Класът `string` съдържа множество методи за обработка на стрингове. В таблица 13.2 са дадени някои от тях.

Таблица 13.2. Прототипи и описания на някои методи от класа `string`.

Прототип	**Описание**
size_t size() const; size_t length() const;	Връща дължината (броя на символите) на стринг.
size_t max_size() const;	Връща максималната дължина (капацитет), която стрингът може да има.
size_t capacity() const;	Връща броя на символите, които могат да бъдат записани в стринга, без да се налага разширяване на капацитета.
void resize (size_t n);	Увеличава капацитета на стринга с n на брой елемента (символа).*
void clear();	Прави стринга да съдържа 0 символа.
bool empty() const;	Връща true, ако стрингът има дължина 0 и false в противен случай.
string& insert (size_t pos, const string& str);	Вмъква допълнителни символи, намиращите се в str, точно преди символа на позиция pos.*
string& erase (size_t pos = 0, size_t len =	Премахва len на брой символа от

npos);	стринга, редуцирайки дължината му, като започва от позиция pos.*
string& replace (size_t pos, size_t len, const string& str);	Заменя символите от позиция pos, обхващайки len на брой, с нови от str.*
void swap (string& str);	Разменя съдържанията на стринга и str.
size_t find (const string& str, size_t pos = 0) const;	Претърсва стринга за първото срещане на str в него. Връща позицията му. Ако е подаден втори параметър, търсенето започва от позиция pos.*
const char* c_str() const;	Копира стринга в масив от символи (C string) и връща указател към него.

* Предефиниран метод.

Типът на данни `size_t` е дефиниран в стандартната библиотека на C++ и е еквивалентен на `unsigned long`.

Ето един пример, състоящ се от програмен текст, който демонстрира употребата на някои от методите, дадени в таблица 13.2:

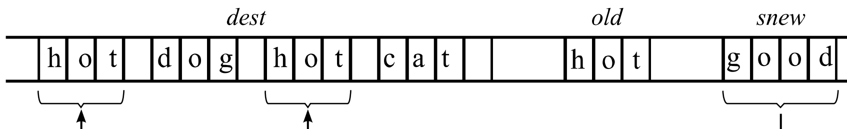
```
string name = "Emiliyan Petkov";
string str = "Georgiev ";
name.insert( 9, str );
cout << name << endl;
cout << "size: " << name.size() << endl;
cout << "length: " << name.length() << endl;
cout << "capacity: " << name.capacity() << endl;
cout << "max_size: " << name.max_size() << endl;

Изход:
Emiliyan Georgiev Petkov
size: 24
length: 24
capacity: 30
max_size: 1073741820
```

Предаване на стрингове като параметри на функции се осъществява по стойност, а не по адрес, както това е при масивите от символи. Ако е необходимо да се направи предаване по адрес, това трябва да се обозначи с *, а предаване по псевдоним – с &.

Задача 13.1. Да се напише функция, която приема като параметри три стринга `dest`, `old` и `snew`. Функцията да замества `old` с `snew` навсякъде, където `old` се среща в `dest`. Функцията да връща стринга `dest`.

Анализ на задача 13.1: Това, което функцията трябва да прави, може да бъде илюстрирано с фигура 13.1:



Фиг. 13.1. Разпределение на стринговете в паметта (по задача 13.1).

Трябва да се търси `old` в `dest` (чрез метода `find()`) и там, където бъде намерен, да се заменя със `snew` (чрез метода `replace()`). Когато `old` или `snew` имат нулева дължина, заместване не трябва да се осъществява. Когато броят на символите на `old` и `snew` е един и същ, тогава всеки символ от `snew` замества съответстващия му в `old`. Когато обаче броят на символите на `old` е по-малък (както това е в примера), тогава още символи трябва да бъдат вмъкнати (с метода `insert()`) в `dest`, а когато броят на символите на `old` е по-голям, тогава символи ще трябва да бъдат премахнати (с метода `erase()`) от `dest` преди копирането на символите на `snew`. Това се налага от факта, че методът `replace()` ще замества толкова символи в `dest` на указаната позиция, колкото е броят на символите в `snew`.

Реализация на задача 13.1:

```
string ReplaceAll( string &dest, string old,
                              string snew )
{
    int old_len = old.length();
    if( old_len > 0 ) {
        int pos = 0;
        int snew_len = snew.length();
        int balance = snew_len - old_len;
        while( (pos = dest.find( old, pos )) !=
                              string::npos ){
            if( balance >= 0 )
```

```
                dest.insert( pos,balance, ' ' );
            else
                dest.erase( pos, -1*balance );
            dest.replace( pos, snew_len, snew );
            pos++;
        }
    }
    return dest;
}
```

Функцията може да бъде извикана в друга функция по следния начин:

```
string d = "hot dog hot cat hot horse";
string o = "hot", n = "good";
cout << ReplaceAll( d, o, n ) << endl;
```

Тестване на програмата: За да е гарантирано, че функцията работи надеждно, тя трябва да бъде тествана със следните комбинации от входни данни:

```
d = "", o = "hot", n = "good";
d = "hot dog hot cat", o = "", n = "good";
d = "hot dog hot cat", o = "hot", n = "";
d = "hot dog hot cat", o = "hot", n = "good";
d = "hot dog hot cat", o = "hot", n = "not";
d = "hot dog hot cat", o = "hot", n = "no";
```

като извежда съответно:

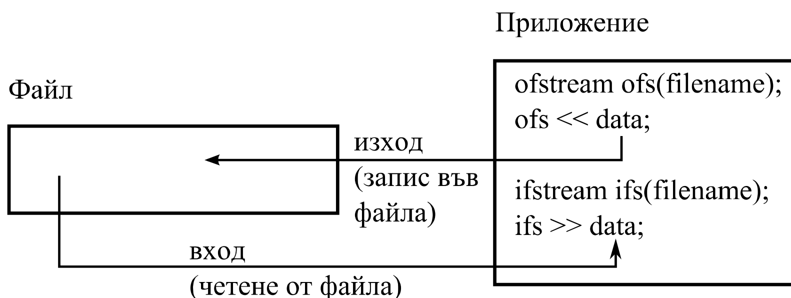
```
      //празен стринг
hot dog hot cat
hot dog hot cat
good dog good cat
not dog not cat
no dog no cat
```

Глава XIV. Файлове

Файловете в компютърните системи съдържат данни. Файловете се съхраняват като последователност от байтове върху носител на информация (твърд диск, компакт диск, флаш памет и др.) [6]. Организацията на файловете се извършва от файлова система.

14.1. Файлови потоци.

В езика C++ записването и четенето на данни в и от файлове се осъществява чрез входни и изходни потоци. Класовете, които са дефинирани за работа с файлове, са: `fstream` – за четене и запис (вход и изход), `ifstream` – за четене (вход), `ofstream` – за запис (изход). Смисълът на процесите четене, запис, вход и изход са уточнени на фигура 14.1.



Фиг. 14.1. Процесите вход и изход.

Свързването на поток с физически файл може да се направи чрез конструктора на съответния клас на файлов поток

```
fstream fs( char* filename, openmode mode );
```

или чрез метода на класа `open`, който има следния прототип:

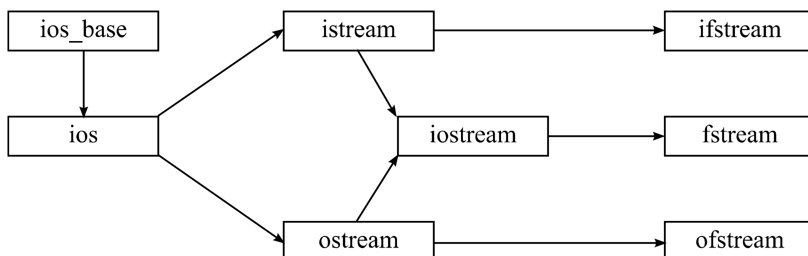
```
void open( char* filename, openmode mode );
```

Параметърът `filename` трябва да бъде символен низ, указващ името и ако е необходимо и директорията на файла. Ако в символния низ директория не е посочена, тогава файлът се търси в папката на проекта (или изпълнимия файл). Вторият параметър `mode` указва режим на отваряне на файла. Режимът може да бъде един или комбинация (свързани с побитова операция „или") от дадените в таблица 14.1.

Таблица 14.1. Режими на входни и изходни файлови потоци.

ios::in	Отваря за входни операции.
ios::out	Отваря за изходни операции.
ios::binary	Отваря в двоичен режим.
ios::ate	Установява началната позиция да бъде в края на файла. Ако този флаг не е установен, началната позиция е в началото на файла.
ios::app	Всички изходни операции се извършват в края на файла, добавяйки новите данни към текущото съдържание на файла.
ios::trunc	Ако файлът е отворен за изходни операции и той вече съществува, неговото предходно съдържание се изтрива и се записва новото.

На фигура 14.2 може да се види мястото на класовете ifstream, ofstream и fstream в йерархията от класове в библиотеката iostream. Много от методите им са наследени. Прототипите на всички методи от тези класове могат да бъдат видени в документацията на C++.



Фиг. 14.2. Класовете ifstream, ofstream и fstream в йерархията от класове в библиотеката iostream.

14.2. Текстови файлове.

Файловете могат да бъдат отваряни като *текстови* (text) или *двоични* (binary). Текстовите файлове са тези, които съдържат ASCII символи, а двоичните – последователност от байтове, които обикновено се групират в обекти, за да може съдържанието им да се интерпретира правилно. Отварянето на файл по подразбиране става в текстов режим. Ако той трябва да бъде отворен в двоичен режим, тогава това трябва да се укаже с `ios::binary`.

Ето един пример на програма, която създава текстов файл и записва кратък текст в него:

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream ofs;
  ofs.open( "example.txt", ios::out );
  ofs << "This a text file.\n";
  ofs.close();
  cout << "File example.txt was created!\n";
  return 0;
}
```

След като всички операции във файла са завършени, той трябва да бъде затворен. Това се прави с метода `close()`. Ако файлът остане отворен след приключване на програмата, тогава съдържанието му се губи и размерът му ще бъде 0 байта.

Ето един пример на програма, която чете от текстов файл и извежда съдържанието му в командния интерпретатор:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main ()
{
  string line;
  ifstream ifs( "example.txt", ios::in );
  if( ifs.is_open())
```

```
  {
    while( getline( ifs, line ) )
    {
      cout << line << '\n';
    }
    ifs.close();
  }
  else cout << "Unable to open file!\n";

  return 0;
}
```

В тази програма се използва методът `is_open()`, който връща `true`, ако файлът е бил коректно отворен, и `false` в противен случай. Функцията `getline(…)` чете от входния поток цял ред от символи (до знак за край на ред, но без него), записва го в обект от `string` и връща стойност `true`, а ако е достигнат краят на файла, т.е. няма символи, които да бъдат прочетени, връща `false`.

Задача 14.1. Да се напише програма, която чете реални числа, записани в текстов файл `numbers.txt` на всеки ред по едно. Програмата да намира сумата на числата, след което да записва тази сума на нов последен ред във файла.

Реализация на задача 14.1.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    double d, Sum = 0.0;
    char fname[] = "numbers.txt";
    fstream fs;
    fs.open( fname, ios::in );
    if( fs.fail() ) {
        cout << "Error opening file: "
            << fname << "!\n";
        return 1;
    }
    while( ! fs.eof() ){
        fs >> d;
```

```
            Sum += d;
        }
        fs.close();
        fs.open( fname, ios::out | ios::app );
        fs << endl << Sum;
        fs.close();
        return 0;
    }
```

В програмата е дефиниран потокът `fs` от `fstream`. Първо този поток се използва, за да се отвори файлът за четене, като се четат числата, записани на всеки ред и се натрупва сумата. След затварянето на файла, чрез същия поток отново се отваря файлът, но за запис на намерената сума.

14.3. Моментни местоположения при четене и запис.

Винаги обектите на входно-изходните потоци съхраняват вътрешно поне по една моментна (достигната) позиция при четене и запис. Това означава, че при четене от файл се съхранява местоположението, до което е стигнало четенето. При следващо четене от файла входният поток ще извлече символите, намиращи се от тази позиция нататък. При запис във файл се съхранява местоположението, до което е достигнало записването. При следващо вмъкване на символи в потока, те ще бъдат записани от тази позиция нататък. Всеки байт във файла е нова позиция, като позицията на първия байт е 0, а следващите се бележат с последователни положителни числа.

Тъй като четенето и записването са отделни операции, моментните местоположения при четене и записване могат да бъдат различни. Следователно обектите за тяхното съхраняване са два – за позиция при четене (get position) и позиция при запис (put position). В потоците имаме четири функции, чрез които можем както да извличаме, така и да задаваме тези стойности. Това са прототипите на функциите:

```
streampos tellg();
seekg(streampos pos);
streampos tellp();
seekp(streampos pos);
```

Методът `tellg()` връща моментната позиция при четене, `seekg(…)` – задава позиция за четене, `tellp()` – връща моментната позиция при запис, `seekp(…)` – задава позиция за запис.

Методите `seekg` и `seekp` могат да бъдат използвани още и със следните прототипи:

```
seekg( streamoff off, ios_base::seekdir way );
seekp( streamoff off, ios_base::seekdir way );
```

Първият параметър задава отместване. Вторият параметър задава относителна позиция, спрямо която да се извърши отместването. Тя може да бъде една от дадените в таблица 14.2.

Таблица 14.2. Относителни позиции при задаване на местоположение за четене и запис.

ios::beg	Отместването се извършва спрямо началото на файла.
ios::cur	Отместването се извършва спрямо текущата позиция.
ios::end	Отместването се извършва спрямо края на файла.

В следващия програмен фрагмент се намира размерът на файл, асоцииран с входния поток `ifs`:

```
ifs.seekg( 0, ifs.end );
int length = ifs.tellg();
ifs.seekg( 0, ifs.beg );
```

Чрез първото извикване на функцията `seekg` местоположението за четене става краят на файла. След това чрез `tellg` се извлича позицията (в `length`) след последния записан байт във файла. Това отговаря на броя на байтовете във файла, тъй като първият винаги се номерира с 0. Второто извикване на `seekg` връща позицията за четене обратно в началото на файла.

14.4. Двоични файлове.

При работа с двоични файлове форматиране на данните обикновено не се прави. Затова операторите за извеждане и въвеждане (<< и >>) не се използват. В този случай са по-подходящи специално разработените за целта методи `read` и `write` от

файловите потоци. Методът `write` е от класа `ostream`, наследен от `ifstream` и `fstream`, а `read` е от класа `istream`, наследен от `ofstream` и `fstream`. Ето техните прототипи:

```
write(const char* s1, streamsize n1);
read(char* s2, streamsize n2);
```

Методът `write` вмъква (записва) първите `n1` символа от масива, сочен от `s1`, а `read` извлича (чете) `n2` символа от потока и ги записва в масива, сочен от `s2`. Следващата програма демонстрира употребата на тези функции като копира съдържанието на един файл (file.bin) в друг (newfile.bin):

```
#include <fstream>
using namespace std;
int main ()
{
  ifstream ifs( "file.bin", ios::binary );
  ofstream ofs( "newfile.bin", ios::binary );

  ifs.seekg( 0, ifs.end );
  long size = ifs.tellg();
  ifs.seekg( 0 );

  char* buffer = new char[ size ];

  ifs.read( buffer, size );
  ofs.write( buffer, size );

  delete [] buffer;
  ofs.close();
  ifs.close();

  return 0;
}
```

Глава XV. Модули. Тестване на програми

15.1. Модули.

Когато програмният текст в една програма е кратък, най-естествено е той да бъде поместен в един файл. Програмите, които написахме дотук, бяха именно такива. Когато обаче се разработва по-голяма програма, било то от един човек или от екип, кодът ѝ може да бъде разпределен в няколко първични файла, представляващи отделни части, наречени още *модули* [7]. Модулите могат да бъдат създавани, редактирани и компилирани поотделно. По този начин разработването на програмата може да се прави поетапно или разпределено между членовете на екипа. Освен това, веднъж създадени, модулите могат да бъдат използвани в различни проекти.

Всяка програма трябва да притежава само една главна функция. Проектът, който се състои от първични файлове, също трябва да отговаря на това условие. Файлът, в който се помества функцията `main`, можем да наричаме *основен първичен файл*. При компилиране на проекта се компилират само тези файлове, които са били променени след последното компилиране.

Модулите могат да съдържат дефиниции на константи, променливи, структури, функции, класове и др. Идеята е, основният първичен файл да може да се възползва от компонентите на модулите. За да е възможно компилирането на основния първичен файл, е нужен механизъм, по който компилаторът да бъде уведомен за наличието на новите компоненти. Това в C++ се прави като за всеки първичен файл се създаде по един заглавен файл, който да бъде включен както в първичния файл, така и в основния първичен файл. За да не се получи двойно вмъкване на хедър файла (веднъж от първичния файл и втори път от основния първичен файл), се използват директивите на препроцесора `#ifndef`, `#define` и `#endif`.

Възможността за работа с модули в езика C++ ще бъде демонстрирана чрез разработването на една програма в интерактивната среда Code::Blocks. Вече добре знаем, че всяка програма, която се разработва в тази среда, трябва да бъде

организирана като проект. Модулите могат да бъдат написани като част от проекта или отделно от него. В случая, когато са направени предварително, те трябва да бъдат добавени към проекта.

Задача 15.1. Да се разработи програма, която дефинира два класа за представянето съответно на обекти от тип час и дата. Програмата да дефинира по един обект от тези класове, задава съответно час и дата и извежда часа и датата в командния интерпретатор. Двата класа да бъдат реализирани в два модула.

Анализ на задача 15.1. Трябва да бъдат създадени два модула – един за класа, представляващ обекти от тип време и един за класа, представляващ обекти от тип дата. Нека първия да наречем Time, а втория – Date. Нека за класа Time направим един хедър файл, който съдържа дефиницията на класа, чието съдържание често се нарича *интерфейс* (от англ. interface) *на класа,* и един първичен файл, който съдържа дефинициите на методите, чието съдържание често се нарича *реализация* (от англ. implementation) *на класа*.

Добре е всеки модул (двойката първичен файл и хедър) да бъде поместен в една папка. В случая папките на класовете могат да носят техните имена (Time и Date). При създаването на хедър файл в Code::Blocks, средата дефинира запазена дума на хедъра (от англ. header guard word). Тази запазена дума е нужна, за да се избегне двойно вмъкване на хедъра. Това се постига като съдържанието на хедъра се дефинира под тази запазена дума.

Можем да създадем папка Time, в която да поместим файловете time.h и time.cpp. При създаването на нов хедър (time.h), средата ни предлага съдържанието му да бъде дефинирано под запазената дума TIME_H_INCLUDED т.е. TIME_H_INCLUDED да представлява цялото съдържание на хедъра. Запазената дума се дефинира при първото вмъкване на time.h чрез `#include`. Затова съдържанието на файла трябва да се помести в конструкцията:

```
#ifndef TIME_H_INCLUDED
#define TIME_H_INCLUDED
...
#endif
```

Класът Time трябва да съдържа три член-променливи – за часовете, минутите и секундите. Класът трябва да има конструктор по подразбиране, като е добре също да напишем конструктор с

параметри, метод, задаващ стойности на член-променливите, методи за извличане на стойностите на член-променливите и метод за преобразуване на часа в символен низ.

Подобни са и разсъжденията за модула, реализиращ класа Date. Можем да създадем една папка с име Date и в нея да поместим файловете date.h и date.cpp, съдържащи съответно интерфейса и реализацията на класа Date.

Реализация на задача 15.1.

Нека да създадем нов проект в Code::Blocks, например с името Time_Date_App. Тогава файловете на проекта ще се поместят в папка с име Time_Date_App. Към така създадения проект ще добавим два модула, реализиращи класовете Time и Date. Файловете на Time нека да поместим в подпапка на Time_Date_App с име Time и файловете на Date – в подпапка на Time_Date_App с име Date. Основният първичен файл в проекта е main.cpp.

Последователно трябва да бъдат създадени двата хедъра и двата първични файла. След това те трябва да бъдат включени към проекта, т.е. да станат част от него. Това в Code::Blocks се прави чрез контекстното меню, появяващо се при щракване с десния бутон на мишката върху името на проекта в мениджъра, като се избере командата Add files.

Ето целия програмен текст на time.h:

```
//time.h - interface of class Time
#ifndef TIME_H_INCLUDED
#define TIME_H_INCLUDED

#include <sstream>
using namespace std;

class Time {
    unsigned short hours;
    unsigned short minutes;
    unsigned short seconds;
public:
    Time();
    Time( unsigned short h, unsigned short m,
            unsigned short s );
    void SetTime( unsigned short h,
      unsigned short m, unsigned short s );
    unsigned short GetHours();
```

```
    unsigned short GetMinutes();
    unsigned short GetSeconds();
    string TimeInString();
};

#endif // TIME_H_INCLUDED
```

Първичният файл time.cpp трябва да вмъква time.h и да съдържа реализацията на класа Time:

```
//time.cpp - implementation of class Time
#include "time.h"

Time::Time()
{
    hours = minutes = seconds = 0;
}
Time::Time( unsigned short h, unsigned short m,
            unsigned short s )
{
    this->SetTime( h, m, s );
}
void Time::SetTime( unsigned short h,
            unsigned short m, unsigned short s )
{
    if( h >= 24 ) hours = 23;
    else hours = h;
    if( m >= 60 ) minutes = 59;
    else minutes = m;
    if( s >= 60 ) seconds = 59;
    else seconds = s;
}
unsigned short Time::GetHours()
{
    return hours;
}
unsigned short Time::GetMinutes()
{
    return minutes;
}
unsigned short Time::GetSeconds()
{
    return seconds;
```

```
}
string Time::TimeInString()
{
    ostringstream convert;
    if( hours < 10 )
        convert << "0";
    convert << hours << ":";
    if( minutes < 10 )
        convert << "0";
    convert << minutes << ":";
    if( seconds < 10 )
        convert << "0";
    convert << seconds;
    return convert.str();
}
```

Ето целия програмен текст на date.h:

```
 //date.h - interface of class Date
#ifndef DATE_H_INCLUDED
#define DATE_H_INCLUDED


#include <string>
using namespace std;

class Date {
    unsigned short day;
    unsigned short month;
    unsigned short year;
public:
    Date();
    Date( unsigned short d, unsigned short m,
      unsigned short y );
    void SetDate( unsigned short d,
      unsigned short m, unsigned short y );
    unsigned short GetDay();
    unsigned short GetMonth();
    unsigned short GetYear();
    string DateInString();
};

#endif // DATE_H_INCLUDED
```

Първичният файл date.cpp трябва да вмъква date.h и да съдържа реализацията на класа Date:

```
//date.cpp - implementation of class Date
#include "date.h"

Date::Date()
{
    day = month = 1;
    year = 2000;
}
Date::Date( unsigned short d, unsigned short m,
            unsigned short y )
{
    this->SetDate( d, m, y );
}
void Date::SetDate( unsigned short d,
            unsigned short m, unsigned short y )
{
    bool leap_year = false;
    if( year <= 1970 ) year = 1970;
    else year = y;
    if( year % 4 == 0 ) leap_year = true;
    if( year % 100 == 0 ) leap_year = false;
    if( year % 400 == 0 ) leap_year = true;
    if( m >= 12 ) m = 12;
    if( m == 0 ) m = 1;
    month = m;
    if( month == 2 )
        if( leap_year )
            if( d >= 29 ) day = 29;
            else day = d;
        else
            if( d >= 28 ) day = 28;
            else day = d;
    else
        if( month == 1 and month == 3
            and month == 5 and month == 7
            and month == 8 and month == 10
            and month == 12 )
                if( d >= 31 ) day = 31;
                else day = d;
```

```
        else
            if( d >= 30 ) day = 30;
            else day = d;
}
unsigned short Date::GetDay()
{
    return day;
}
unsigned short Date::GetMonth()
{
    return month;
}
unsigned short Date::GetYear()
{
    return year;
}
string Date::DateInString()
{
    ostringstream convert;
    if( day < 10 )
        convert << "0";
    convert << day << ".";
    if( month < 10 )
        convert << "0";
    convert << month << ".";
    convert << year;
    return convert.str();
}
```

Ето и един кратък вариант на основния първичен файл:

```
#include <iostream>
using namespace std;

#include "time.h"
#include "date.h"

int main()
{
    Time T;
    T.SetTime( 24, 65, 4 );
    Date D( 30, 2, 2008 );
    cout << T.TimeInString() << ", "
```

```
                << D.DateInString() << endl;
    return 0;
}
```

15.2. Тестване на програми.

Тестването на една програма е процес, при който се търсят логически грешки в нея. Тези грешки много често жаргонно се наричат бъгове (от англ. bugs) [1]. Затова и този процес е получил прозвището дебъгване (от англ. debugging).

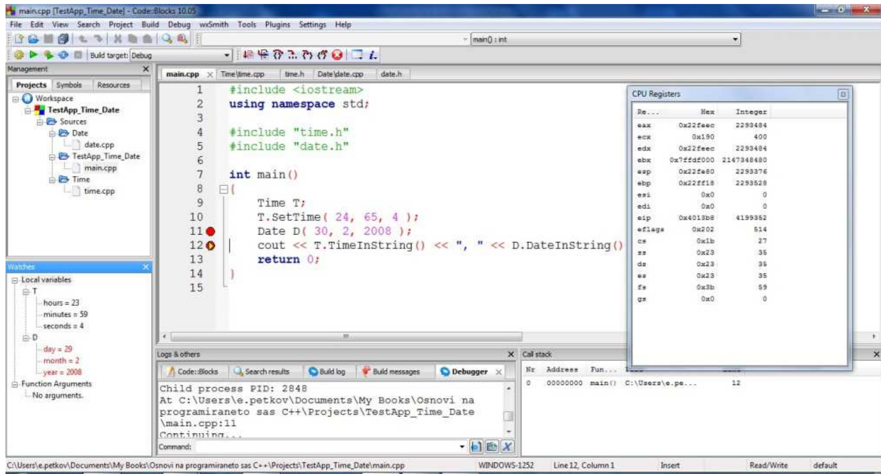
Основните задачи при тестването са: намиране на проблем в работата на програмата, откриване на местоположението на грешката в програмния текст и нейното отстраняване. Когато програмата е модулна, тестването се осъществява като се тества всеки модул поотделно, след което отново се тества цялата програма.

Всяка съвременна интерактивна среда за разработка на програми притежава средство, с помощта на което по-лесно да се откриват логически грешки. Това са така наречените *дебъгери* (от англ. debuggers). Работата на дебъгерите зависи от това, което се тества и от конкретната операционна система.

Обикновено дебъгерите изискват да знаят процеса, който ще се наблюдава. Това се прави като се подаде или името на процеса или неговия идентификатор. При тестване на програма в интерактивна среда това се прави автоматично.

При тестване с дебъгер, програмата може да бъде изпълнявана стъпка по стъпка (ред по ред и/или до позицията на курсора в програмния текст), като в това време се наблюдават стойностите на променливи и обекти, както и стойностите на регистрите на централния процесор. Друга възможност е поставянето на точки на прекъсване (от англ. break points) в програмния текст. Те се поставят за определени редове. В този случай дебъгера изпълнява програмата до достигането на такава точка. Бъде ли достигната точка на прекъсване, дебъгера спира изпълнението на програмата, за да позволи наблюдението на моментни стойности на променливи, указатели, обекти и др. След това програмата отново може да бъде пусната. Изпълнението ѝ продължава до достигането на следваща точка на прекъсване или завършване. Работата на програмата в режим на тестване може да бъде прекратена чрез команда на дебъгера по всяко време.

В средата Code::Blocks командите за работа с дебъгера се намират в менюто Debug. На фигура 15.1 е демонстрирана работата на дебъгера като са поставени две точки на прекъсване, отворени са прозорците за наблюдение на стойности на компоненти на програмата (Watches) и на регистрите на централния процесор (CPU Registers) като програмата е спряна точно преди изпълнението на 12-ти ред. Така могат да се видят стойностите на член-променливите на двата обекта (T и D), дефинирани в `main`.



Фиг. 15.1. Работа на дебъгера в Code::Blocks.

Литература

1. Donaldson Scott E., Stanley G. Siegel. Successful Software Development. Prentice-Hall, Inc., 2001. 745 p.
2. McConnell Steve. Code Complete, Second Edition. Microsoft Press, 2004. 960 p.
3. Stroustrup Bjarne. An Overview of the C++ Programming Language. The Handbook of Object Technology, CRC Press LLC, Boca Raton, 1999. 25 p.
4. Stroustrup Bjarne. Programming - Principles and Practice Using C++. Addison-Wesley, 2008. 1236 p.
5. Stroustrup Bjarne. The C++ Programming Language, Third Edition. Addison-Wesley, 1997. 1022 p.
6. Симов Георги. Програмиране на С++. ДБ „Абагар", Велико Търново, 1993. 298 стр.
7. Хорстман Кай. Принципи на програмирането със С++. ИК „Софтех", София, 2000. 729 стр.

ЕМИЛИЯН ПЕТКОВ

ОСНОВИ НА ПРОГРАМИРАНЕТО СЪС C++

ПЪРВО ИЗДАНИЕ

ISBN:

Коректор: Радостина Петкова
er.petkovi@gmail.com

Издателство: „Фабер”
тел. (062) 600 650
www.faber-bg.com

Велико Търново, 2014

View publication stats