ЕЛЕНА ЗАХАРИЕВА-СТОЯНОВА

ПРОГРАМИРАНЕ И ИЗПОЛЗВАНЕ НА КОМПЮТРИ

ръководство за лабораторни упражнения

2021

Катедра: **Компютърни Системи и Технологии**
Дисциплина: **Програмиране и Използване на Компютри**

Ръководството за лабораторни упражнения „Програмиране и използване на компютри“ включва лабораторните упражнения по едноименната дисциплина, включена в учебния план на специалност „Компютърни Системи и Технологии“ (КСТ) на факултет „Електротехника и електроника“ (ЕЕ) на Технически Университет - Габрово. Изданието се използва и при провеждане на лабораторните упражнения по дисциплината „Въведение в програмирането“, включена в учебния план на специалност „Софтуерно и компютърно инженерство“ (СКИ) на факулет ЕЕ при ТУ-Габрово. Дисциплините „Програмиране и използване на компютри“ и „Въведение в програмирането“ имат около 90% съвпадение. Разликата е, че в дисциплина „Въведение в програмирането“ е заложено изучаването единствено на програмен език С.

Ръководството може да бъде използвано от студенти от други специалности, както и от специалисти за обучение по програмиране на език С/С++.

ПРОГРАМИРАНЕ И ИЗПОЛЗВАНЕ НА КОМПЮТРИ

Ръководство за лабораторни упражнения

второ преработено и допълнено издание

Рецензент: доц. д-р инж. Росен Стефанов Иванов

ISBN 978-954-683-658-8

Лабораторно упражнение № 1
Алгоритмизация на изчислителни процеси

I. Цел на лабораторното упражнение

Целта на упражнението е запознаване на студентите с видовете алгоритми, създаване на практически умения за разработка и документиране на алгоритми на програмни задачи. Специално внимание се отделя на документирането на алгоритмите според изискванията на съществуващите нормативни документи.

II. Теоретична обосновка

1. Понятие за алгоритъм. Видове алгоритми

Алгоритъм е основно понятие в изчислителната техника. Интуитивно се определя като ***съвкупност от указания (инструкции, действия, команди), при изпълнението на които се получава търсеният резултат***.

Според реда на изпълнение на действията, алгоритмите се разделят на:

- линейни;
- разклонени;
- циклични.

Линейни са тези алгоритми, при които се спазва нормалният, линейно последователен ред на изпълнение на действията (операциите). При линейните алгоритми, действията се изпълняват по реда на тяхното записване.

При ***разклонените алгоритми*** се нарушава линейният ред на изпълнение на операциите т.е. действията не се изпълняват по реда на тяхното записване. В разклонените алгоритми, според дадено условие, в даден момент се изпълнява една част от действията. В последствие, при друго изпълнение на алгоритъма, при други условия, ще се изпълни друга част от действията.

Циклични са тези алгоритми, при които част от действията се повтарят многократно. Това многократно повторение се нарича ***цикъл***. Като структура, цикълът се състои от: ***условие за край*** и ***тяло*** (групата операции, които се повтарят многократно).

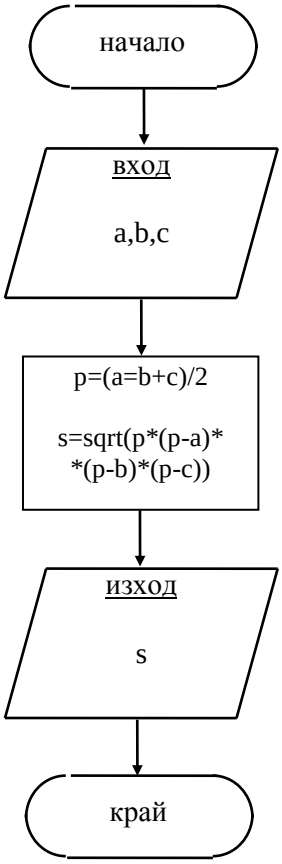
В зависимост от това, дали условието за край е преди тялото или след него, циклите се разделят на:

- цикъл с пред условие;
- цикъл със след условие (пост условие).

При цикъл с предусловие, условието за край е преди тялото, докато при цикъл със следусловие, условието е след тялото на цикъла. Основната разлика между двата вида цикли е, че *докато цикълът с предусловие е възможно никога да не се изпълни, ако още в началото условието е изпълнено, цикълът със след условие ще бъде изпълнен поне веднъж*.

2. Линейни алгоритми

Линейни алгоритми се използват когато е необходимо да бъдат извършени група изчисления. Обикновено такива алгоритми са или част от друг алгоритъм или алгоритми на подпрограми, които използват редица пресмятания. Пример на програмна задача с линеен алгоритъм е: *Да се намери лицето на триъгълник по зададени три страни, като се счита, че входните данни са въведени коректно.* Алгоритъмът на задачата е даден на фиг. 1.1.



Фиг. 1.1. Пример за линеен алгоритъм

3. Разклонени алгоритми

В разклонените алгоритми, според зададено условие се извършва ***разклонение*** към един или към друг клон от алгоритъма. Типичен пример за задача с разклонен алгоритъм е намирането на корените на квадратно уравнение по зададени стойности на коефициентите.

Квадратното уравнение има следния общ вид:

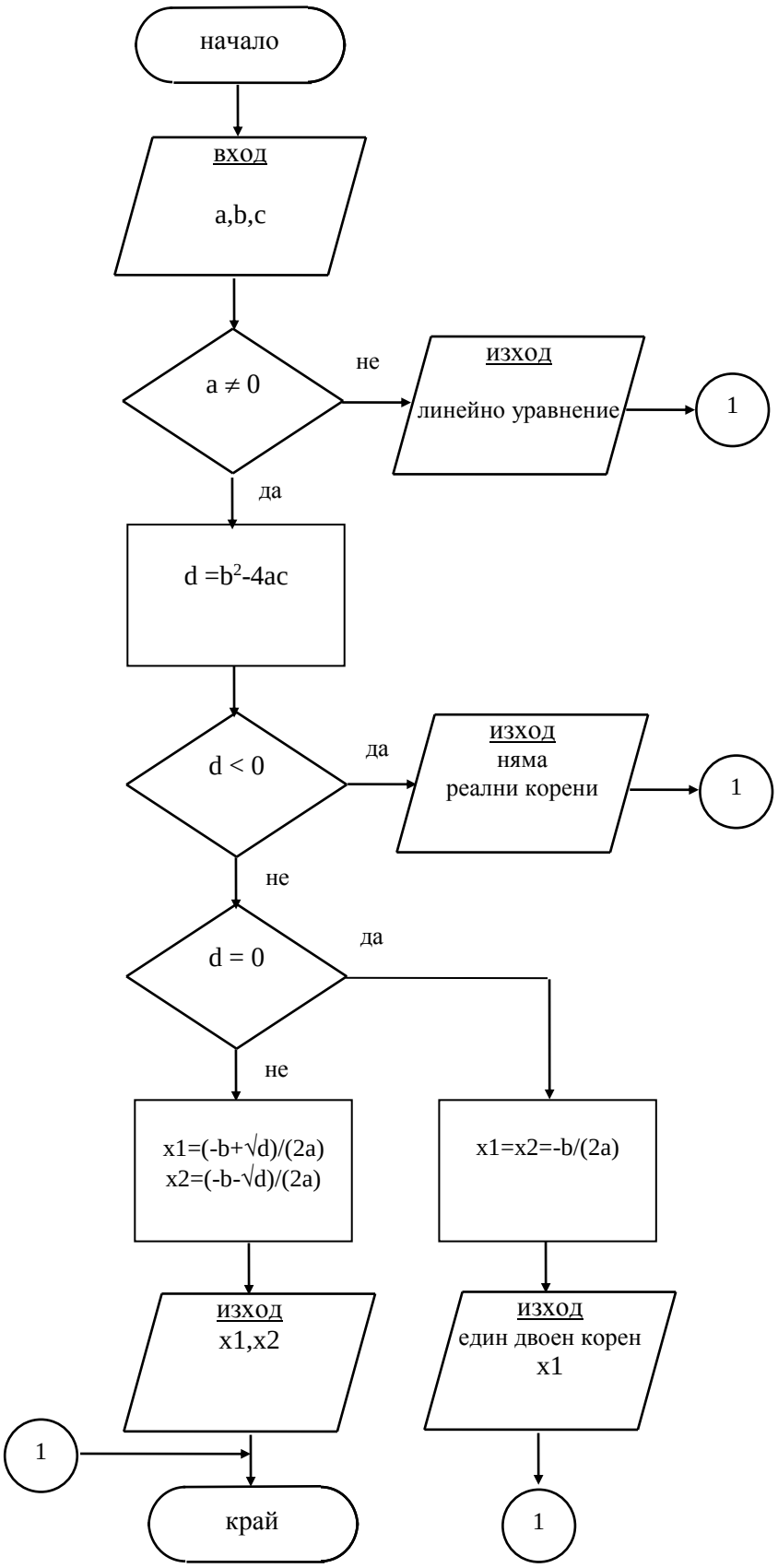
$ax^2+bx+c=0$

където a, b, c се наричат коефициенти на квадратното уравнение

Намирането на корените на уравнението е по формулата:

$$x_{12} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Величината подкорена се нарича дискриминанта и се означава с d (т.е $d=b^2-4ac$). За да има дадено уравнение реални корени, дискриминатата трябва да е по-голяма или равна на нула. Освен това, за да е квадратно уравнението, стойността на a трябва дае различна от 0. Съответно, алгоритъмът трябва да съобрази тези условия като се направят необходимите проверки. Алгоритъмът на задачата за намиране корените на квадратно уравнение е даден на фиг. 1.2.

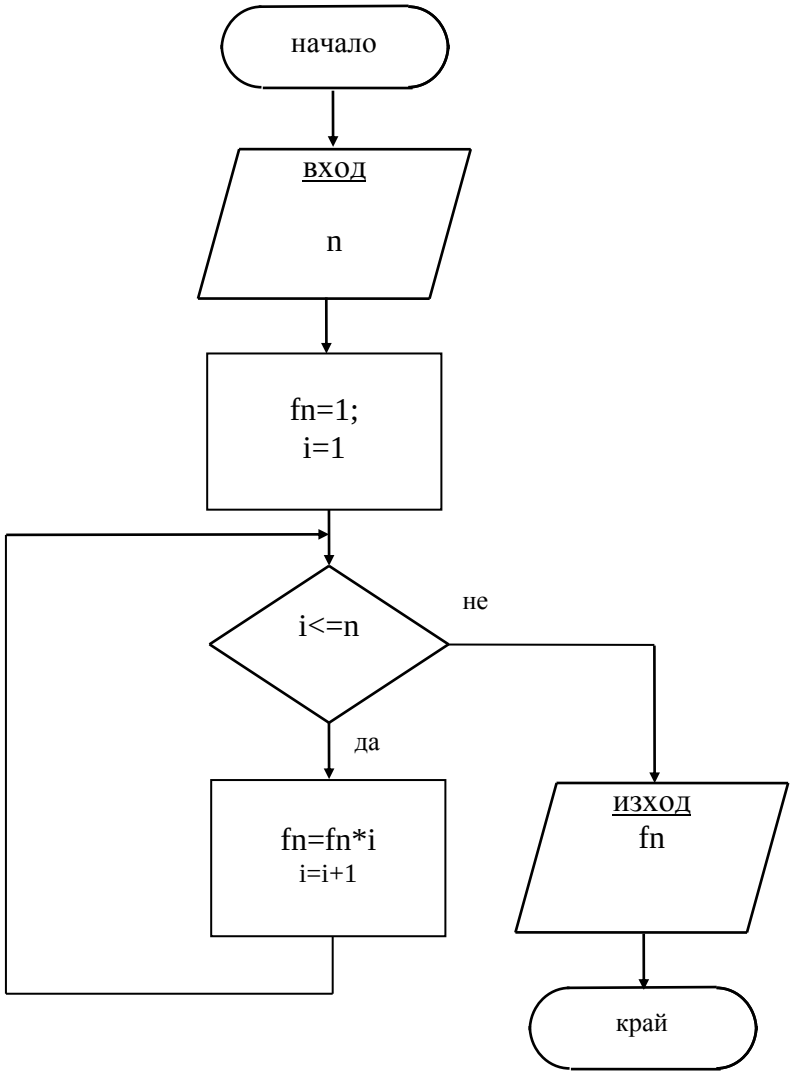


Фиг.1.2. Пример за разклонен алгоритъм

4. Циклични алгоритми

При цикличните алгоритми част от действията се повтарят многократно. Тези действия съставляват т.нар. **тяло на цикъл**. Важна част от всеки един програмен цикъл е условието за край. Чрез него се определя при какво условие да се прекрати изпълнението на многократно повтарящите се действия. Ако условието за край липсва или ако е зададено некоректно, тогава цикълът се превръща безкраен.

На фиг. 1.3. е представен пример за цикличен алгоритъм - задачата за намиране на n! (n факториел). Стойността на n се въвежда от клавиатурата. Този алгоритъм е типичен пример за цикличен алгоритъм.

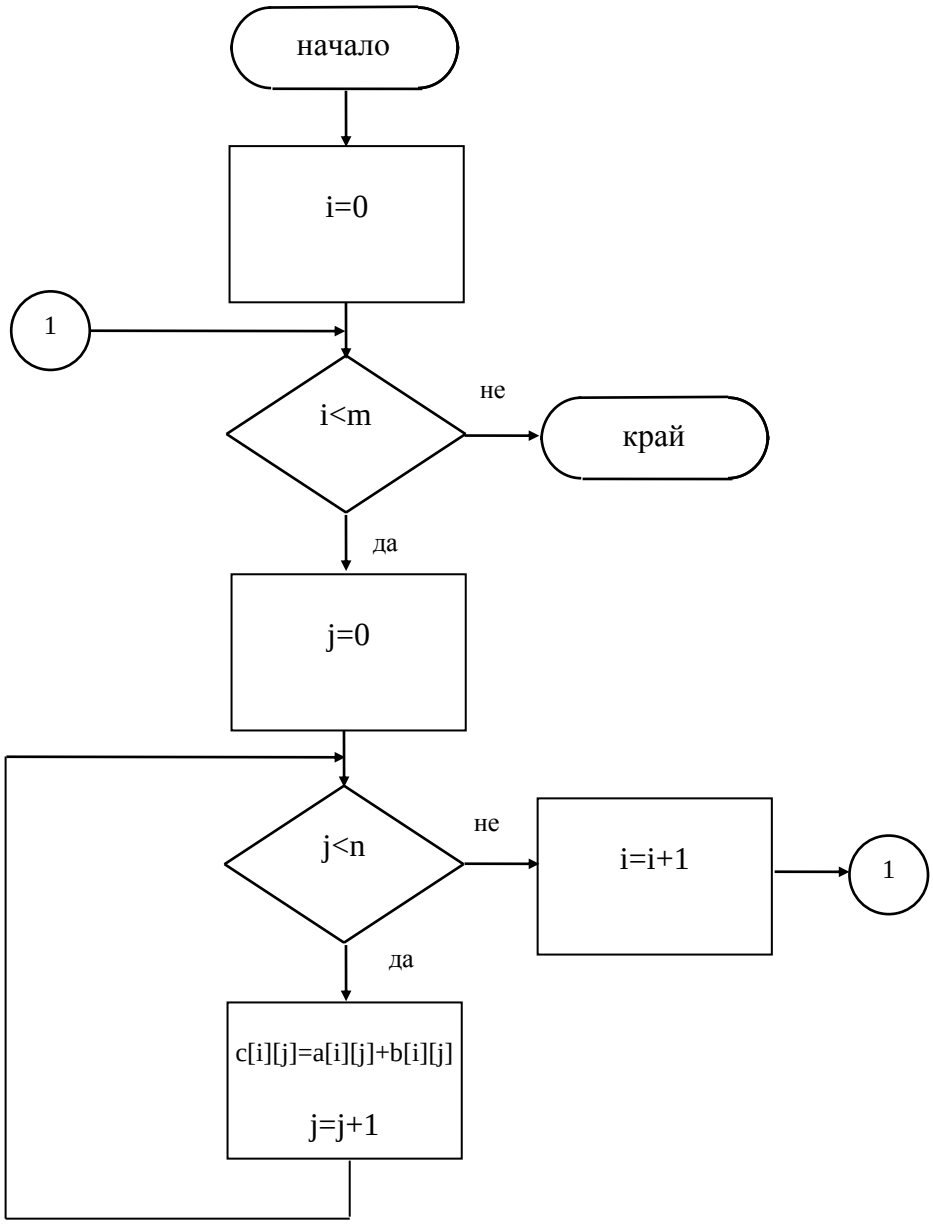


Фиг. 1.3. Пример за цикличен алгоритъм

Много често при реализацията на циклични алгоритми се срещат т.нар. ***вложени цикли***. В този случай, в тялото на даден цикъл се организира друг цикъл. Пример за такъв алгоритъм е представения на фиг. 1.4, чрез него се реализира задачата за сумиране на две матрици. (При програмната реализация матрицата се представя като двумерен масив. Всеки елемент от масива има два индекса, определящи неговата позицията в ред и колона на матрицата). Входни матрици са A, B; изходна матрица е C. Счита се, че елементите на матриците са предварително въведени. Стойностите за m, n определят броя на редове и стълбове, съответно. Индексите на елементите се изменят както е според C и C++, от 0 до m-1 и от 0 до n-1. Цикълът на изменение на елементите по стълбове, реализиран с изменението

на брояча j, е вложен в цикъла на изменение на редовете, реализиран с изменението на брояча i. При тази реализация, цикълът на изменение на редовете е ***външен***, а цикълът на изменение на стълбовете е ***вътрешен***.

Важно условие за реализацията на вложени цикли е те да не се пресичат т.е. цикълът, който е започнал първи, да свърши последен.



Фиг. 1.4. Пример за цикличен алгоритъм с вложени цикли

III. Контролни въпроси

1. Кои алгоритми са линейни?
2. Кои алгоритми са разклонени?
3. Кои алгоритми са циклични?

4. Каква е разликата между цикъл с пред условие и цикъл със след условие?

5. Кои цикли са вложени?

IV. Задачи за изпълнение

1. Да се състави алгоритъм за намиране на обем на правилна триъгълна пирамида. Стойностите на страните на основата и височината се задават от клавиатурата.

2. Да се състави алгоритъм за въвеждане на две числа и намиране на по-голямото от тях.

3. Да се състави алгоритъм за намиране стойностите на функцията y по зададен аргумент x:

y=2, при x<=0;
y=x+2, при x ϵ (0,1);
y=3, при x ϵ [1,2];
y=5-x, при x ϵ (2, 3);
y=2, при x>=3;

4. Да се състави алгоритъм за намиране на произведението на произволно въведени положителни числа от клавиатурата. Въвеждането на стойност 0 да прекрати понататъшното въвеждане на числа.

5. Да се състави алгоритъм за намиране позицията и стойността на минималния елемент в едномерен масив от 10 елемента.

6. Да се състави алгоритъм за транспониране на матрица. В алгоритъма да се включи и въвеждането на елементите на матрицата.

Лабораторно упражнение № 2
Структура на C/C++ програма. Среда за програмиране. Въвеждане и извеждане на данни.

I. Цел на лабораторното упражнение

Целта на упражнението е да бъдат запознати студентите със средата за програмиране MS Visual Studio и основните етапи от развитие на програмите; да бъдат създадени навици и умения за редактиране, компилиране, свързване и изпълнение на програми, като се обърне внимание на откриването и премахването на грешките в програмите. В упражнението се акцентира и на общата структура на програма на C и C++.

II. Теоретична обосновка

1. Структура на програма на език C

Всяка програма, написана на език C се състои от отделни програмни единици, наречени *функции*. Броят на функциите в програмата е неограничен, но винаги трябва да има една функция, дефинирана като главна – това е функция **main()**.

Функция main() осъществява връзката с операционната система. При стартиране на изпълнимия модул (.COM или .EXE), операционната система предава управлението на главната функция main(). Съответно тази функцията може да извиква други функции т.е. да им предава управлението. След проключване на своето изпълнение, всяка функция връща управлението там, където е извикана. Главната функция приключва последна като връща управлението в Операционната система (ОС).

Функция е набор декларации и последователно написани оператори и синтактични конструкции на езика, осъществяващи действия, които представляват логическа цялост и реализират определен алгоритъм.

Общата структура на програма, написана на език C може да се представи чрез следните компоненти:

```
#include <io.h>          // описания на заглавни файлове
#include <stdio.h>

int number;              // дефиниции на външни променливи
unsigned char byte;


 main()                  // главна функция
{
  ...                    // тяло на главната функция
}
// функция 1
...
// функция 2
```

Като пример за програма, написана на език C++ може да се представи решението на задачата за намиране на сумата на елементите от едномерен масив:

```
#include <stdio.h>
void main()
```

```
{   int a[10];
   int i, suma;
      printf("въведи стойности за елементите на масива:\n")
   for (i=0;i<20;i++)
     {printf("a[%d]",i);
      scanf("%d", &a[i]);
      }
   suma=0;
   for (i=0;i<20;i++) suma+=a[i];
   printf("сумата е:%d", suma);
}
```

2. Етапи от развитието на програмите

Процесът на създаване на програмно осигуряване преминава през следните основни етапи:

2.1. Създаване и редактиране на първичния код

На този етап, въз основа на разработения алгоритъм, се създава първичния (***source***) код. Това е програмата написана на алгоритмичен език. Първичният код се записва като ASCII файл, като обикновено разширението подсказва какъв е използвания програмен език. За език С, разширението на файла е **.C**, а за език С++, разширението е **.CPP**.

2.2. Компилиране

На етап ***компилиране*** първичният код на програмата, написан на алгоритмичен език, се превежда на машинен език. Програма-транслатор, която осъществява превода на първичния код се нарича ***компилатор***.

Компилаторът анализира първичния код и когато открие синтактични грешки или несъответствия, издава съответните съобщения. В този случай, е необходимо първичният код да бъде редактиран отново, с цел премахване на грешките, и повторно компилиран. При успешно компилиране, се създава т.нар. ***обектен код*** – програмата, преведена на машинен език. Обектният код се записва във файл с име, което съвпада с името на първичния код, но с разширение **.OBJ**.

Компилаторите позволяват да се промени името на файла с обектния код т.е. имената на файловете с първичния и обектния код могат да не съвпадат. Обикновено промяната на името на обектния код рядко се практикува.

2.3. Свързване

На етап ***свързване***, обектният код се свързва със системните библиотеки и други обектни модули. Програмата се прави изпълнима под управлението на операционна система. Свързването се осъществява с програма ***свързващ редактор (linker)***.

При успешно приключване на етапа, свързващият редактор създава т. нар. ***изпълним модул,*** записан като изпълним файл за съответната операционна система. За операционна система WINDOWS, изпълнимите файлове са с разширение **.COM** или **.EXE**.

При условие, че на етап свързване възникнат грешки, необходимо е първичният код да бъде редактиран и повторно компилиран.

2.4. Тестване и настройка

Въпреки, че след етапа свързване, програмата е готова за изпълнение, това не означава, че тя работи коректно. Необходимо е програмата да бъде тествана дали при съответните входни данни се получават верни резултати.

В голяма част от случаите, първоначално разработеното програмно осигуряване не работи коректно. Грешките се дължат на логическо несъответствие с алгоритъма или на неверен алгоритъм. В етапа ***тестване и настройка*** се откриват и отстраняват логическите грешки в програмата. Разработени са специални програми за тестване и настройка, които се наричат ***дебъгери***. Дебъгерите използват следните средства за тестване на програми:

- постъпково изпълнение на програмата;
- задаване на точки на прекъсване;
- наблюдение на променливи.

3. Среда за програмиране Microsoft Visual Studio.

За създаването на програмно осигуряване, като е започне със създаването на първичния код и се приключи с тестване на изпълнимия модул, са необходими следните системни програми: ***текстов редактор*** за редактиране на първичния код, ***компилатор, свързващ редактор и дебъгер***. В програмните системи, тези програми са обединени в една обща, интегрирана, **развойна програмна среда**. Развойната среда включва както средства за създаване и редактиране на текстови файлове, така и средства за компилиране, свързване, тестване и настройка на програмно осигуряване. MS Visual Studio е мощна интегрирана развойна среда, чрез която се изграждат различни видове приложения. Тя поддържа множество програмни езици (C#, C++, VB), както и различни технологии за разработка на софтуер (Win32, COM, ASP.NET, ADO.NET, Windows Forms, WPF и др.). MS Visual Studio предоставя възможности за писане, компилиране, изпълнение и тестване на програмен код, изграждане на потребителски интерфейс, моделиране на данни и много други функции.

В настоящото ръководство се разглежда създаването на конзолни приложения. Конзолата е прозорец на ОС, чрез който потребителите взаимодействат със системните програми или с други конзолни приложения. Чрез конзолата потребителят въвежда данни от стандартното входно устройство (кавиатура) и получава изход от програмата на стандартното изходно устройство (екран).

3.1. Създаване на конзолни приложения чрез Microsoft Visual Studio.

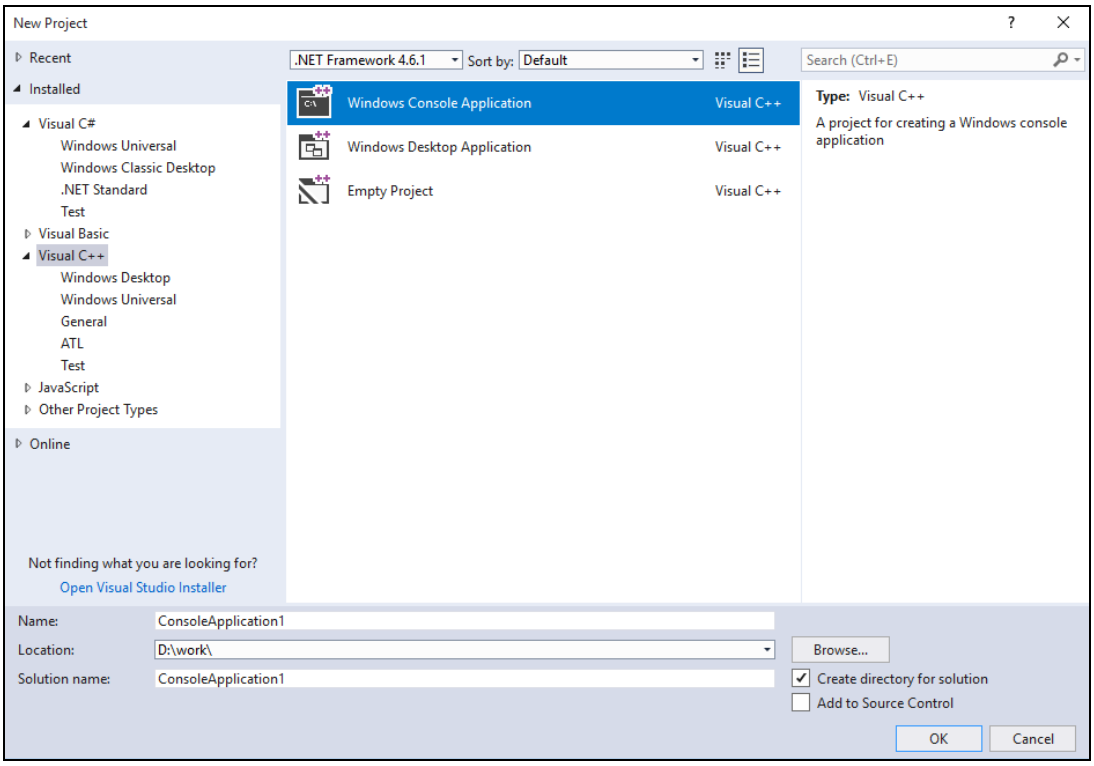
Когато се започне с изграждането на дадено приложение, първо се създава **проект.** Чрез проектът се групират множеството файлове, които изграждат една цялостна програмна система. За начинаещи програмисти е препоръчително всяка една отделна задача, колкото и елементарна да е, да се оформя като самостоятелен програмен проект.

Създаването на проект е със следните стъпки:

- Команда File/New Project и се отваря диалог New Project (фиг.2.1).
- Избира се език за програмиране, в случая C++. Да се има предвид, че средата за програмиране MS Visual Studio поддържа няколко програми езика, освен C++.
- Избира се тип на проекта, в случая конзолно приложение (**Windows Console Application**).

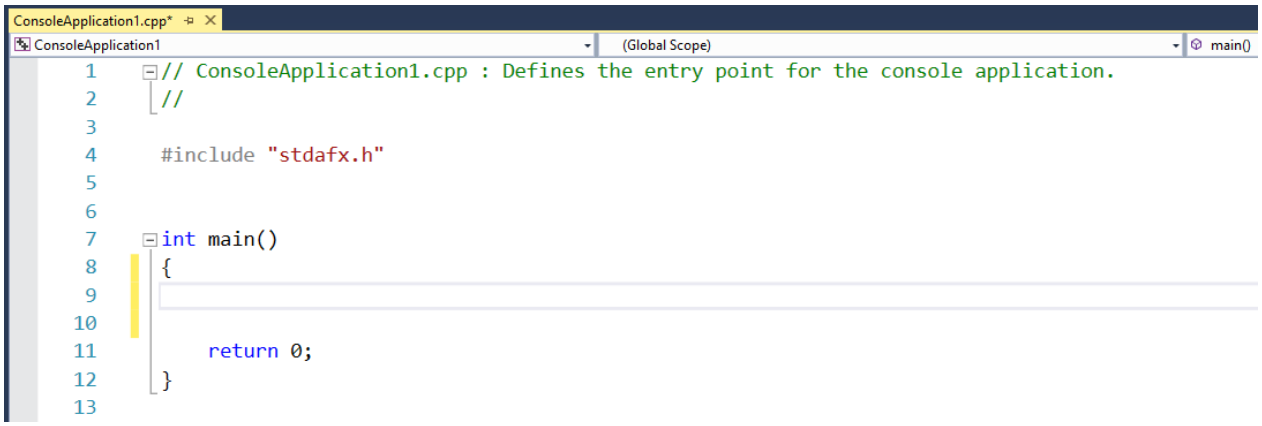
Конзолните приложения са програми, чрез които входът и изходът е на **системната конзола** (клавиатурата и екрана) т.е. те очакват вход от клавиатурата и отпечатват като текст на екрана изходен резултат. Този тип приложения са удобни за начално обучение по програмиране.

- Определя се името на проекта (поле Name) и папката в която ще се съхранява (поле Location). Изборът на дисково устройство и папка е чрез бутон Browse



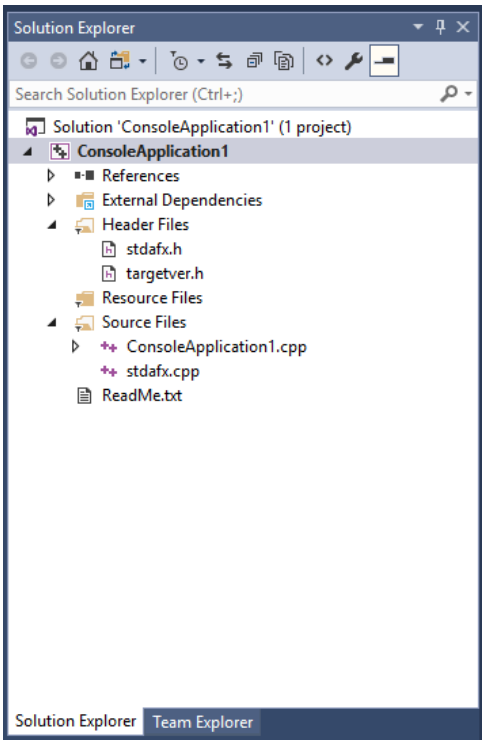
Фиг. 2.1. Създаване на нов проект с Visual Studio

При създаването на нов проект MS Visual Studio генерира програмен код, който формира т.нар. **скелет на приложението**. Генерира се и дефиницията на функцията **main()**, в чието тяло програмистът може да добави съответния програмен код. (фиг.2.2)



Фиг. 2.2. Програмен код на функция main

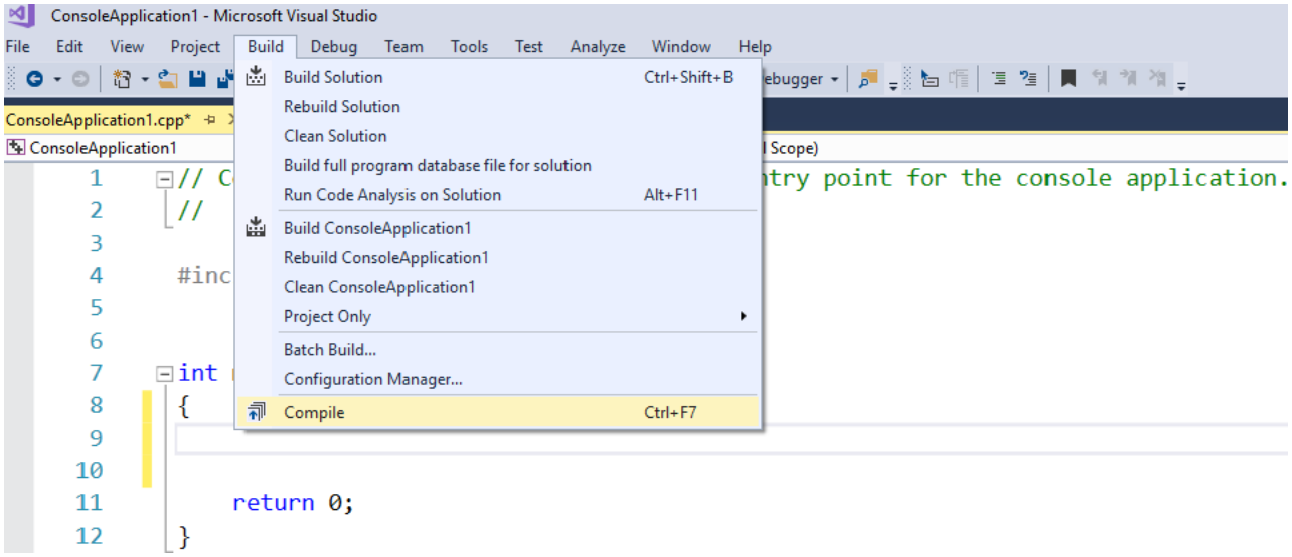
Програмният код се съхранява във файл с име, определено от програмиста и разширение **.cpp.** Освен този, са създадени още няколко файла, които съставляват текущия проект. Структурата на проекта т.е. файловете, които го изграждат могат да се видят и да се управляват чрез т.нар. Solution Explorer (фиг. 2.3).



Фиг. 2.3. Solution Explorer

3.2. Компилиране и свързване на проекта

За да се компилиране и свързване на програмния код се използват група команди от основното меню **Build** (фиг.2.4). Чрез команда **Build/Compile** се компилира програмния код. Свързването е чрез команда **Build/Build (име на проект)** или чрез **Build/Build Solution**. Първата свързва само файловете в текуция проект, докато втората свързва всички проекти в текущия контейнер (solution). Ако програмният код не е компилиран, последните две команди стартират първо компилатора, а след това свързващия редактор.



Фиг. 2.4. Команди за компилиране и свързване на програмния код

Ако компилаторът открие синтактични грешки, програмният код не може да бъде компилиран и съответно, на екрана се извеждат откритите при компилацията грешки (фиг. 2.5). Необходимо е програмистът да редактира програмния код и отново да го компилира. Това се повтаря докато не бъде създаден първо обектен, а в последствие и изпълним код.

Фиг. 2.5. Списък на откритите при компилиране грешки

3.3. Стартиране

За да се стартира изпълнението на програмния код се натиска клавишна комбинация Ctrl +F5. Така след изпълнението на програмния код екранът на конзолата не се затваря, а се изчаква да се натисне произволен клавиш. На екрана се изписва съобщението “Press any key to continue…” . Ако изпълнението на програмният код се стартира чрез клавиш F5, след приключване на изпълнението, конзолата се затваря автоматично.

III. Контролни въпроси

1. Какво представлява функция в език C ?
2. Какво е предназначението на програма ***компилатор***?
3. Какво е предназначението на програма ***свързващ редактор***?
4. Какво представлява понятието ***source (първичен)*** код?
5. С кои команди се извършва ***компилиране*** на първичния код?
6. Кои приложения се определят като конзолни?
7. Посочете стъпките за създаване на проект в среда на MS Visual Studio.
8. С кои команди се компилира и свързва програмен код в среда на MS Visual Studio?

IV. Задачи за изпълнение

1. Да се разгледа средата за програмиране Visual C++.
2. Да се тества примерната програмата за намиране сумата от елементите на едномерен масив.
3. Да се открият и отстранят синтактичните грешки в програмата.

Лабораторно упражнение № 3
Аритметични типове данни. Въвеждане и извеждане на данни.

I. Цел на лабораторното упражнение

Целта на упражнението е запознаване на студентите с функциите за въвеждане и извеждане на данни при работа с конзолни приложения, аритметичните типове данни, дефинирането на константи и променливи в език C++.

II. Теоретична обосновка

1. Аритметични типове данни

Основна характеристика на всеки език за програмиране са типовете данни, които поддържа. Типът на константа или на променлива определя нейното съдържание т.е. от какъв вид е информацията: числов, символен, цели или реални числа и т.н. Основни типове данни в C/C++ са *аритметичните типове*: ***int, char, float, double***.

Тип ***char*** служи за представяне на символни данни (константи и променливи). Размерът на типа е 1 байт (8 бита), което означава, че стойността на дадена променлива или константа от тип **char** е в границите на [-128, +127], понеже този тип е за знакови числа. При представяне на знаково число с 8 бита, най-старшият бит се използва като бит за знак: 0 за знак + и 1 за знак -, а останалите 7 бита са за стойността на числото. При беззнаковите числа, за стойността на числото се използват всичките 8 бита. Числената стойност на променлива от тип ***char*** е ***ASCII кода*** на символа.

Тип ***int*** е предназначен за представяне на цели числа със знак. Обхвата на типа зависи от настройките на компилатора: за 16 бита (1 дума) е [-2^{15}, $+2^{15}-1$], докато при 32 бита (1 двойна дума) е [-2^{31},$+2^{31}$].

Тип ***float*** служи за представяне на реални (дробно-десетични) числа. Обхвата на променлива от тип ***float*** при 32-битов размер е [±10E-37, ± 10E+37].

Чрез тип ***double*** се представят реални (дробни) числа с двойна точност. При размер на типа 64 бита, обхвата на променлива от тип ***double*** е [±10E-307, ± 10E+307].

Аритметичните типове могат да бъдат използвани със следните ***модификатори***:

- ***signed*** - за знакови числа;
- ***unsigned*** - за беззнакови числа;
- ***short*** - за числа с единична точност;
- ***long*** - за числа с двойна точност.

Модификаторите променят (модифицират) типа на числото. Необходимо е да се отбележи, че не всеки модификатор може да се приложи към всеки тип. Например, типовете за дробно-десетични числа ***float, double*** не могат да бъдат беззнакови; също – модификатори ***short*** и ***long*** не могат да бъдат прилагани към тип ***char***.

2. Променливи и константи. Дефиниции

Променливите са данни, чиито стойности се изменят по време на изпълнение на програмата, докато *константите* са данни, които не се променят по време на изпълнение на програмата. Използваните в програмата данни трябва да бъдат дефинирани. Дефиницията е необходима за заделяне на памет за съответната данна.

Общият вид на дефиницията на променлива е следният:

тип име [=стойност][, име =[стойност],…];

Задължителни елементи на дефиницията са името и типа на променливата. Едновременно, дефинирането на променлива може да бъде съпроводено и с инициализация т.е. със задаване на начална стойност на променливата.

На един ред могат да бъдат дефинирани повече от една променливи, които имат един и същи тип.

Примери за дефиниране на променливи:

```
unsigned j;          // дефиниране на променлива j от тип unsigned int
int x,y;             // дефиниране на променливи x,y от тип int
char ch='Y';         // дефиниране на променлива ch с тип char и начална
                     // стойност 'Y'
float pi=3.14;       // дефиниране на променлива pi с тип float и начална
                     // стойност 3.14
unsigned char h;     // дефиниране на променлива h от тип unsigned char
int z=2,k;           // дефиниране на променливи z с начална стойност 2 и k
                     // от тип int
```

Подобно на променливите, константите имат тип, обозначават се чрез символни имена (идентификатори), но стойностите им са постоянни. След дефиниране на константа, при всяко нейно срещане в програмата, компилаторът заменя символното име със стойността й. Ако, в процеса на разработка на програмата се наложи промяна на параметър, определен чрез константа, единствената корекция в тази програма е замяна стойността на една константа. Ако, не е използвана константа, а само стойност, в този случай, ще се наложат корекции навсякъде в програмата, където се среща параметърът.

В език C++ съществува начин за дефиниране на константи чрез използване на ключова дума ***const***. Общият вид на дефиницията е следния:

const [тип] име=стойност;

Примери:

```
const int N=100;
const float PI=3.14;
```

Ако типът на константата не се зададе явно, използва се стойността й за да се зададе тип по подразбиране:

```
const y=3;           // типът на константата е int
const z=0.8          // типът на константата е float
```

В език C не съществува ключова дума ***const***. За дефиниране на константи в C се използва директива на предпроцесора **#define,** чрез която символното име се заменя с посчената сойност. Примери:

```
#define  N      100
#define  PI     3.14
#define  Y      3
```

3. Въвеждане и извеждане на данни

3.1. Функции printf() и scanf()

В C/C++ не са предвидени оператори за вход и изход, тъй като по правило те са машиннозависими. Вместо тях, са използвани функции за вход и изход, разпространявани като стандартна библиотека, заедно с компилатора на езика. Най-често използваните функции за въвеждане и извеждане на данни са: ***scanf()*** и ***printf().*** Тъй като тези функции се считат за ненадеждни, след версия 2013 на Visual Studio се препоръчва вместо тях да се използват функции ***scanf_s()*** и ***printf_s()***.

Функциите ***scanf()*** и ***printf()*** са декларирани в ***stdio.h***, което означава, че за да бъдат използвани е необходимо да включване на заглавния файл:

#include <stdio.h>

Функция ***printf()*** служи за извеждане на данни и има следната декларация:

int printf("форматиращи параметри", списък_аргументи);

Форматиращите параметри са комбинация от: *низова константа*, която се извежда на екрана във вида, в който е записана и *низ форматни спецификации*, които не се показват на екрана, но служат за управление начина на извеждане на аргументите. В низовата константа могат да са включени и управляващи символи. Например '\n' - символ за нов ред.

Аргументите са променливи, константи или изрази. Функцията printf() извежда на екрана стойностите на аргументите, според зададените форматни спецификации. Всеки аргумент има свой форматиращ параметър т.е. броя на форматиращите параметри е равен на броя на аргументите.

Форматиращите параметри започват със символ **%**, последвани от модификатори и спецификации. Видовете форматни спецификации са:

%d – за десетично цяло число със знак;

%u – за десетично цяло число без знак;

%o – за цяло осмично число;

%x – за цяло шестнадесетично число, като се използват символи a, b, c ,d ,e, f;

%X - за цяло шестнадесетично число, като се използват символи A, B, C ,D ,E, F;

%f – за дробно-десетично число от тип float в F формат; (за дробно-десетично число от тип double в F формат е необходимо да се добави модификатор **l** т.е. видът на спецификацията е **%lf**)

%e – за дробно-десетично число в експоненциален (E) формат, като се използва символ e;

%E - за дробно-десетично число в експоненциален (E) формат, като се използва символ E;

%g – за дробно-десетично число в E или в F формат, като се използва символ e;

%G - за дробно-десетично число в E или в F формат, като се използва символ E;

%c – за единичен символ;

%s – за низ.

Заедно с форматните спецификации, във форматиращите параметри могат да бъдат добавени и различни модификатори. По този начин, общият вид на форматиращия параметър може да се представи като:

% [флагове][поле][.точност] [F\N\h\l\L] форматна_спецификация

където:

флагове са следните символи:

- задава ляво изравняване;

+ задава знак + пред положителните числа;

интервал положителните стойности започват с интервал, вместо с знак +;

определя добавяне на символ 0 пред осмичните числа и 0x или 0X пред шестнадесетичните.

поле – определя минималния брой на символите за извеждане;

точност – определя минималния брой символи след десетичната точка; използва се само за дробно-десетични числа;

F|N|h|l|L- определят дължината на аргумента, като:

N – близък указател;

F – далечен указател;

h – тип ***short int***;

l – тип ***long***;

L – тип ***long double***.

Примери:
printf("I like C and C++."); //отпечатва низ

float x,y;
…
printf("Резултатът е %f", y); //отпечатва стойността на y

int a=3,b=8;
…
printf ("a+b=%d",(a+b)); //отпечатва резултата от сбора на a и b

Въвеждането на данни е чрез функция ***scanf().*** Функцията има следната декларация:
int scanf("форматиращи параметри", списък_аргументи);

Форматните спецификации са същите, както при функция ***printf()***.

Като аргументи се използват адресите на променливите, чиито стойности се въвеждат. Съответно, за целта се използва адресен оператор &, който извлича адреса на посочената. Функцията ***scanf()*** връща броя на успешно обработени входни символи.

Примери:
int k;
printf("input value of k=");
scanf("%d", &k);

float r;
printf(" r=");
scanf("%f", &r);

3.2. Стандартни потоци cin и cout.

Освен чрез стандартните функции за въвеждане и извеждане на данни, в C++ е предвидена библиотека ***iostream***, при която стандартния вход и изход се поддържа от два потока данни: ***cin*** за вход и ***cout*** за изход.

Потокът за стандартен вход (***cin***) се насочва към клавиатурата, а потока за стандартен изход (***cout***) – към екрана

Потокът ***cout*** служи за извеждане на данни на екрана и се използва съвместно с предефинирания оператор **<<.**

Оператор << се нарича още ***оператор за вмъкване*** Чрез него се вмъкват намиращите се след него данни за потока. Данните могат да бъдат: *низови константи; числени и символни константи; променливи; изрази*.

Примери:
cout<<"Въведи стойност за r="; // извежда на екрана дадения низ
cout<<100; // извежда на екрана стойност 100

cout<<x; *// извежда на екрана стойността на променливата x*

Операторът за вмъкване може да се използва повече от един път в един и същ израз. Пример:

cout<<"Резултатът е: "<<y<<'\n';

В случая, '\n' е управляваща константа за нов ред.

Преминаването на нов ред може да се използва и манипулатор ***endl*.**

Пример:

cout<<"Резултатът е: "<<y<<endl;

Работата със стандартния входен поток ***cin*** се осъществява с помощта на оператора за извличане **>>**. Оператор >> трябва да бъде следван от променлива, която ще съхранява прочетената стойност.

Потокът ***cin*** може да обработва входа от клавиатурата само след натискане на клавиш Enter т.е. дори и да е въведен даден символ, докато не бъде натиснат клавиш Enter, символът не се обработва.

Пример:

int x;
…
cin>>x;

Стойността, която се въвежда от клавиатурата трябва да съответства на типа на променливата. Несъответствието между типовете на променливите и въведените стойности води до поява на грешки.

Стандартния входен поток може да се използва и за въвеждане на стойности на повече от една променлива. Пример:

float a,b;
…
cin>>a>>b;

// записът е еквивалентен на:

float a,b;
…
cin>>a;
cin>>b;

За да бъдат въведени или изведени цели числа в шестнадесетична бройна система като се изпозват стандартните потоци ***cin*** и ***cout*** може да се използва модификатор ***hex*.** Например:

Int n;
Cout<<"Enter a hexadecimal value:";
Cin>>hex>>n;
Cout<<"The value is: "<<hex<<n;

За въвеждане и извеждане на числа в осмична бройна система се използва модификатор ***oct*.**

III. Контролни въпроси

1. Дефинирайте целочислени променливи x, y.

2. Дефинирайте реални променливи с двойна точност p, q.

3. Дефинирайте константи PI=3.14 и N=100.Какъв е типът на константите?

IV. Задачи за изпълнение

1. Да се създаде конзолно приложение, с което да се представят възможностите на функциите ***printf_s()*** и ***scanf_s()*** за въвеждане и извеждане на данни от различен тип.
2. Да се създаде конзолно приложение за преобразуване на цяло число от десетична в шестнадесетична и осмична бройна система и обратно, като се използват форматните спецификации %d, %X, %x, %o.
3. Да се създаде конзолно приложение, с което да се представят възможностите на стандартните входен и изходен поток за въвеждане и извеждане на данни.

Лабораторно упражнение № 4
Аритметични оператори. Реализация на задачи с линеен алгоритъм. Стандартни математически функции. Преобразуване на типове

I. Цел на лабораторното упражнение

Целта на упражнението е да бъдат запознати студентите със стандартните математически функции; да се създадат навици и умения за използването им при писане на програми на C/C++; да бъде разгледано преобразуването на типове, като се обърне внимание на възможни евентуални грешки от преобразуването на типове.

II. Теоретична обосновка

Линейни алгоритми са тези, които запазват нормалната линейна последователност на изпълнение на операциите т.е. операциите се изпълняват по реда на тяхното записване. В програмните системи линейните алгоритми не се срещат в чист вид, а само като част от друг алгоритъм.

Линейните алгоритми обикновено се използват за математически изчисления. За програмна реализация на линейни алгоритми са необходими: пресмятане на аритметични изрази; оператор за присвояване; функции за въвеждане и извеждане на данни.

1. Аритметични оператори

Език С поддържа унарни и бинарни оператори. Бинарните използват от два операнда, докато в унарните участва само един операнд. Бинарни оператори са:

- ***събиране (+);***
- ***изваждане (-);***
- ***умножение (*);***
- ***деление (/);***
- ***деление по модул (%).***

Унарни оператори са:

- ***унарен + (запазване на знака на числото);***
- ***унарен - (промяна на знака на числото);***
- ***инкрементиране ++ (увеличаване на стойността с 1);***
- ***декрементиране -- (намаляване на стойността с 1).***

2. Изрази. Оператор за присвояване

Израз е правило за изчисляване на стойност т.е. всяка валидна за езика комбинация операции и операнди (данни), по която се пресмята стойност. В резултат на изчислението на даден израз се получава стойност.

Синтаксисът на оператора за присвояване е:
променлива = израз;

Действието на оператора е**: пресмята се стойността на израза отдясно и се присвоява на променливата отляво**.

В език С съществува ***съкратена форма на оператора за присвояване***:
операнд1 ***операция*****= операнд2;**
Тази форма е еквивалента на:
операнд1 = (операнд1) ***операция*** **(операнд2);**

Например, вместо *p=p+2;* записът е *p+=2;*

3. Стандартни математически функции

Тъй като аритметичните операции в програмните езици се ограничават до основните математически операции, за решаване на по-сложни математически задачи и извършване на изчисления в програмите, към компилаторите се добавят библиотеки със стандартни математически функции. Такива са функциите за: пресмятане на квадратен корен, повдигане на степен, пресмятане на логаритъм, тригонометрични изчисления и др.

При компилаторите на C/C++, стандартните математически функции са дефинирани в заглавен файл ***math.h***. Затова, когато се използват тези функции в програмите, се включва заглавния файл:

#include <math.h>

Най-често използвани стандартни, математически функции са:

- ***sqrt()*** – пресмята квадратен корен от посочения в скобите аргумент

Функцията има следната декларация:

double sqrt(double x);

Фукнцията sqrt() изисква като аргумент константа, променлива или израз от тип double и връща резултат от същия тип. Ако аргумента не е от посочения тип, компилаторът се опитва да преобразува типа до double.

- ***pow()*** – използва се за повдигане на степен;

Функцията има следната декларация:

double pow(double x, double y);

Функцията връща като резултат, стойността на израза: $\mathbf{x^y}$. Аргументите са от тип double и функцията връща резултат от същия тип.

- ***exp()*** – пресмята стойността на израза $\mathbf{e^x}$;

Функцията има следната декларация:

double exp(double x);

Резултатът от функцията е пресмятане на израза $\mathbf{e^x}$, като **x** е аргумент на функцията. Аргументът и резултатът от функцията са от тип ***double***.

- ***log()*** - пресмята стойността на израза **ln x**;

Функцията има следната декларация:

double log(double x);

Резултатът от функцията е пресмятане на израза **ln x**, като **x** е аргумент на функцията. Аргументът и резултатът от функцията са от тип ***double***.

- ***log10()*** - пресмята стойността на израза **lg x**;

Функцията има следната декларация:

double log10(double x);

Резултатът от функцията е пресмятане на израза **lg x**, като **x** е аргумент на функцията. Аргументът и резултатът от функцията са от тип ***double***.

- ***abs(), fabs()*** – функциите пресмятат модул от зададения аргумент;

Функциите има следните декларации:

int abs(int x);
double fabs(double x);

Действието на двете функции е идентично – пресмятат абсолютната стойност от зададения аргумент **x**. Разликата между двете функции е, че ***abs()*** изисква аргумент от тип ***int***

и връща резултат от същия тип, докато функция ***fabs()*** изисква аргумент от тип double и връща резултат от тип ***double***.

- ***sin()*** – пресмята синус от зададения ъгъл.

Функцията има следната декларация:
double sin(double x);
Функцията пресмята **Sin(x)**, като аргументът **x** е ъгъл, зададен в радиани. Аргументът и резултатът от функцията са от тип ***double***.

- ***cos()*** – пресмята косинус от зададения ъгъл.

Функцията има следната декларация:
double cos(double x);
Функцията пресмята **Cos(x)**, като аргументът **x** е ъгъл, зададен в радиани. Аргументът и резултатът от функцията са от тип ***double***.

- ***tan()*** – пресмята тангенс от зададения ъгъл.

Функцията има следната декларация:
double tan(double x);
Функцията пресмята **tan(x)**, като аргументът **x** е ъгъл, зададен в радиани. Аргументът и резултатът от функцията са от тип ***double***.

4. Преобразуване на типове

Преобразуването на типове е често срещана операция в програмните езици. Преобразуването на типове е или ***явно***, чрез използване на специален оператор, или ***неявно*** – автоматично, по съответните правила. Явното преобразуване на типове е само в език C++.

4.1. Неявно преобразуване на типове

Неявното преобразуване на типове се осъществява автоматично в следните случаи:

- при изчисляване на изрази;
- при присвояване на стойност;
- при връщане стойности от функции.

Автоматичното преобразуване на типове не винаги е възможно. В случай, има несъответствие на типовете и не е възможно тяхното преобразуване, компилаторът издава съобщение за грешка.

4.2. Явно преобразуване на типове *(typecasting)*

Явното преобразуване на типове е чрез оператор ***привеждане (typecast)***.
Синтаксисът на оператора е:
(тип)израз;

Действието на оператор ***typecast*** е: привежда типа на израза към типа, указан в скобите.

III. Контролни въпроси

1. Напишете програмни редове, с които да пресмятате следните изрази:

$$y=\sqrt{\frac{2a-b}{a+b}}-\sqrt{a^2-b^2};\ y=\sqrt{\frac{2x}{y+9}}+\sqrt{\frac{x-y}{2x}};\ y=\frac{x-k}{4xk}+\frac{1}{2k};\ \frac{a}{a^2+b^2}+\frac{b}{2a}$$

2. Какво ще се изведе на екрана след изпълнението на следния програмен код?

```
int x,y,z;
x=10;
y=10;
z= ++x - y++;
printf("\n%d %d %d",x,y,z);
```

3. Какво ще се изведе на екрана след изпълнението на следния програмен код?

```
double d=8.0/3.0;
printf("\nd=%10.3lf",d);

d=8/3;
printf("\nd=%10.3lf",d);
```

4. Какво ще се изведе на екрана след изпълнението на следния програмен код?

```
int i, j, k;
i = 3;
j =2*(i++);
k =2*(++i);
printf("\n%d %d %d",i,j,k);
```

IV. Задачи за изпълнение

1. Да се състави програма за размяна на две числа.
Упътване: Задачата може да бъде решена по два начина:
- чрез използване на междинна променлива;
- без използване на междинна променлива.

2. Да се състави програма за преобразуване на температура от скалата на Фаренхайд в скала на Целзий.

Упътване: Формулата за преобразуване е: $C = \frac{F - 32}{1.8}$,

където C и F са температура по Целзий и Фаренхайт, съответно.

3. Да се състави програма за намиране на размерите на следните типове: int, char, float, double, short, long, signed, unsigned, unsigned char, long double.
Упътване: Да се използва оператор sizeof.

4. Да се състави програма за намиране на лице на триъгълник по зададени три страни. Стойностите на страните се задават от клавиатурата.

5. Да се състави програма, чрез която се извършва ротация на точка с координати x, y на определен ъгъл на завъртане. Стойностите на координати на точка и ъгъла на завъртане се въвеждат от клавиатурата.

Упътване: Ротацията на точка е графична трансформация, при която точката се завърта под определен ъгъл спрямо началото на координатната система. Новите координати на точката (x1, y1) се пресмятат по формулата:

$x1 = xCos\ \alpha - ySin\ \alpha$

$y1 = xSin\ \alpha + yCos\ \alpha$

като α е ъгълът на завъртане.

6. Да се състави програма за пресмятане на функцията $y=|x^5+\ln x-\lg x|$

Лабораторно упражнение № 5
Побитови оператори

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е запознаване на студентите с приложението на побитовите операциите в C/C++. Операциите с отделни битове са свързани с тази част от езика, която е ориентирана към ниското ниво и се извозват предимно за управление на апаратни устройства. Упражнението е насочено към подготовка за използване на побитовите операции в бъдещи специализирани приложения.

II. Теоретична обосновка

В C и C++ е предвидена възможност за реализиране на операции с отделните битове на операндите чрез т.нар. **поразредни** (**побитови**) оператори. Побитовите оператори могат да се прилагат само върху данни от тип ***int*** и неговите модификации ***short, long, unsigned***.

1. Преместване наляво

Поразрядното преместване наляво с определен брой позиции се осъществява чрез оператор **<<.** Освободените разряди се допълват с нули (0).

Примери:

```
x=0x4f;          // x=0100 1111b
x=x<<2;          // резултатът е: x = 1 0011 1100b =  0x13c

                 // всяко преместване с 1 разряд наляво е
y=1;             // равносилно на умножение по 2
y=y<<1;          // y=010b=2
y=y<<1;          // y=100b=4
```

2. Преместване надясно

Поразрядното преместване надясно с определен брой позиции се осъществява чрез оператор **>>.** Освободените разряди се допълват с нули (0).

Примери:

```
x=0xf8;          // x=1111 1000b
x=x>>2;          // резултатът е: x = 0011 1110b =  0x3e

                 // всяко преместване с 1 разряд надясно е
y=4;             // равносилно на целочислено деление на 2
y=y>>1;                  // y=010b=2
y=y>>1;          // y=001b=1
```

3. Поразрядно И

Оператор поразрядно И е **&**. Резултатът от изпълнението на оператора е представен чрез таблица 1, като x и y са отделни битове.

Таблица 12.1

x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

Побитов оператор И се използва при нулиране на определени битове. Например, нека да бъде нулиран 5 бит на даден 16 разряден регистър.

Стойността (маската), с която ще се нулира бит 5 от 16-разрядният регистър е: 1101 1111 (0xdf).

```
mask=0xdf;          // mask= 1101 1111b
reg=0xf8;           // reg=1111 1000b
reg=reg & mask;     // резултатът е reg = 1101 1000b (0xd8)
```

4. Поразрядно ИЛИ

Оператор поразрядно ИЛИ е | . Резултатът от действието е представен чрез таблица 2, като x и y са отделни битове.

Таблица 12.2

x	y	x|y
0	0	0
0	1	0
1	0	0
1	1	1

Побитова оператор ИЛИ се използва при установяване в 1 на определени битове. Например, нека да бъде установен в 1, бит 5 на даден 16 разряден регистър. Маската е: 0010 0000 (0x20).

```
mask=0x20;          // mask=0010 0000b
reg=0x88;           // reg=1000 1000b
reg=reg | mask;     // резултатът е reg = 1010 1000 (0xa8)
```

5. Поразрядно изключващо ИЛИ

Оператор поразрядно изключващо ИЛИ е ^ . Резултатът от изпълнението на оператора е представен чрез таблица 3, като x и y са отделни битове.

Таблица 12.3

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

Побитов оператор изключващо ИЛИ се използва когато е необходимо да бъдат инвертирани определи битове. Например, нека да бъде инвертиран бит 5 на даден 16 разряден регистър. Маската е: 0010 0000 (0x20).

```
mask=0x20;              // mask=0010 0000b
```

```
reg=0x88;                  // reg=1000 1000b
reg=reg ^ mask;            // резултатът е reg = 1010 1000 (0xa8)
reg=reg ^ mask;            // резултатът е reg = 1000 1000 (0x88)
```

6. Инвертиране на битове

Оператор инвертиране на битовете на операнда е **~**. Резултатът е представен чрез таблица 4.

Таблица 12.4

x	~x
0	1
1	0

Пример:

```
 reg=0x88;          // reg=1000 1000b
 reg1=~reg;         // резултатът е reg1 = 0111 0111b=0x77
```

III. Контролни въпроси

1. Какво ще се изведе на екрана след изпълнението на следния програмен код?

```
int i=5;
int j=10;
printf("\n%d %d %d",i&j,i|j, i^j);
```

2. Какво ще се изведе на екрана след изпълнението на следния програмен код?

```
int i=Ox5;
int j=Ox10;
printf("\n%d %d %d",i&j,i|j, i^j);
```

IV. Задачи за изпълнение

1. Да се създаде конзолно приложение, чрез което се демонстрира действието на побитовите оператори << и >> като се въведе определено число и се преместят битовете определен брой пъти наляво и надясно. Да се използва извеждане на изходния резултат като десетично число със знак, десетично число без знак и шестнадесетично число.

2. Да се създаде конзолно приложение, чрез което се демонстрира действието на побитовите оператори &, |, ^, ~ като се въведат един или два операнда. Да се използва извеждане на изходния резултат като десетично число със знак, десетично число без знак и шестнадесетично число.

3. Да се създаде конзолно приложение, чрез което да се въведе стойност и да се нулира определен бит от числото; да се установи в 1 определен бит и да се промени стойността на определен бит (от 1 в 0 и обратно).

Упътване:
Да се използват побитовите оператори: << за да се премести 1 в определената позиция; | за установяване на бит в единица; ^ за инвертитане на определен бит; ~ за инвертиране на всички битове; & за нулиране на бит.

Лабораторно упражнение № 6
Управляващи конструкции. Реализация на разклонени алгоритми

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е да се запознаят студентите с управляващите конструкции ***if***, ***if-else***, ***switch***; да се създадат умения за програмна реализация на задачи с разклонени алгоритми.

II. Теоретична обосновка

При ***разклонените алгоритми*** се нарушава нормалният, линейно-последователен ред на изпълнение на операциите в програмите т.е. операторите не се изпълняват по реда на тяхното записване в програмата. Реализацията на разклонени алгоритми е с помощта на т.нар. ***управляващи конструкции***. Чрез тях се задава реда на изпълнение на операторите и конструкциите, които формират алгоритъмът на програмата. Доброто познаване на управляващите структури е предпоставка за написване на добре структурирани програми.

1. Конструкция if

Чрез конструкция if се прави избор от два възможни клона на програмата.
Синтаксисът е следният:
if (условие) оператор;

Условието е израз, който се изчислява. Резултатът от изчислението е или TRUE, всяка стойност, различна от 0; или FALSE – стойност равна на 0. Стойността TRUE се интерпретира като вярно твърдение, а стойност FALSE – като невярно твърдение.

Условието може да е променлива или аритметичен израз, не само сравнение или логически израз.

Действието на конструкцията е следното: Пресмята се условието. Ако резултатът е TRUE, изпълнява се операторът след условието. При невярно твърдение, се продължава със следващият оператор след ключова дума ***if.***

2. Конструкция if-else

Служи за избор на една от две възможни алтернативи, в зависимост от стойността на зададен условен израз.

Синтаксис:
if (условие) оператор 1;

else оператор 2;

Действието на конструкцията е следното: Пресмята се условието. Ако стойността на израза е различна от 0 (TRUE), изпълнява се оператор 1. Ако стойността на израза е равна на 0 (FALSE), изпълнява се оператор 2.

3. Конструкция за избор switch

Чрез оператор switch се осъществява избор от няколко възможни варианта.
Синтаксис:

```
switch (израз)
{
  case   константен израз1: оператор 11;оператор 12;...….; break;
  case   константен израз 2: оператор 21;оператор 22;.....; break;
  ..............
  case   константен израз n: операtр n1;оператор n2;……..; break;
```

default : оператор 1; оператор 2;.....;
}

Действието на конструкция ***switch*** е следното: Пресмята се стойността на израза в скобите и се сравнява с константните изрази. Ако има съвпадение на стойностите, изпълняват се операторите след съответния константен израз. Например, нека стойността на израза да е равна на константен израз 1. В този случай, се изпълняват операторите 11, 12, и т.н. докато не бъде срещнат оператор ***break*** или не бъде достигнат края на конструкцията ***switch***. Затова, за да се избегнат неточности в алгоритъма, последният от групата оператори след константите, трябва да е ***break***.

При конструкция ***switch*** е предвидена възможност при несъвпадение с нито една от константите, да се изпълнят група оператори след ключова дума ***default***. Default–частта в оператор switch не е задължителна. Ако липсва default-част в конструкцията switch и стойността на израза не съвпада с нито една от константите, не се изпълнява нито един от операторите в switch.

4. Конструкция break

Конструкция ***break*** предизвиква завършване на итерационните концтрукции (***while***, ***do-while***, ***for***) или на конструкция ***switch***.

За правилната реализация на алгоритми чрез конструкция ***switch***, се използва ***break***. Това се гарантира, че след изпълнение на един от вариантите, останалите няма да бъдат проверявани т.е. ще бъде избран само един от възможните варианти.

Синтаксис:
break;

5. Реализация на разклонения чрез *оператор за условен израз* (*?:*)

Език С разполага с редица полезни оператори, които повишават ефективността на програмирането. Пример за това е операторът за условен израз.

Синтаксис:
операнд 1 ? операнд 2: операнд 3;

Операндите могат да са:константи, променливи, изрази. Операторът изисква 3 операнда. Действието е следното: Пресмята се операнд 1. Ако резултатът е различен от 0 (т.е. резултатът е TRUE), изчислява се резултатът от операнд 2 и неговата стойност е резултатът от оператора. Ако резултатът е от операция 1 е равен на 0 (т.е. резултат FALSE), изчислява се операнд 3 и стойността му е резултат от оператора.

Пример:

```
...
int a,b,c;
...
c=(a>0)?a:b;    /* ако a>0, на c се присвоява стойността на a, в противен случай, на c се присвоява стойността на b */
```

6. Примери за реализация на разклонени алгоритми

Пример 1:

```
// програма за пресмятане корените на квадратното уравнение

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

```
int main()
{
   int a,b,c;           // коефициенти на квадратното уравнение
   double d,x1,x2;      // дискриминанта, корени на квадратното уравнение

   printf("\n a=");
   scanf("%d",&a);
   printf(" b=");
   scanf("%d",&b);
   printf(" c=");
   scanf("%d",&c);

   if(a)                                                    // проверка дали
уравнението е  линейно
    { d=(b*b+4*a*c);                        // проверка за отрицателна
дискриминанта
      if(d<0) printf("Няма реални корени");
      else
       if(d)                                                // проверка дали
дискриминантана е 0
           { x1= -b/(2*a);
             printf("\n Един двоен корен x=%f",x1);
            }
        else
           {x1=(-b+sqrt(d))/(2*a);
            x2=(-b-sqrt(d))/(2*a);
            printf ("\n Корение на уравнението са:%f, %f", x1, x2);
            }
     }
   else printf("Линейно уравнение!");
 }
```

Пример 2:

Пресмятане на стойността на функция y=f(x), според зададена стойност на параметър k

Задачата е: да се състави програма за пресмятане на функцията y като:
$y=x^2+2x+16$, при k=1
$y=\ln x+2x$, при k=2
$y=x^5+2x^3+12$, при k=3

Програмата е пример за използване на оператор ***switch***

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

main()
  {  int k;
      float x;
     double y;
     printf("Enter the value of  k 1 - 3:");
     scanf("%d",&k);
     printf("Enter the value of  x:");
     scanf ("%f",&x);
     switch (k){
                case 1: y=(x*x)+(2*x)+16;
                            printf ("the result is  y=%f",y);
                            break;

                 case 2: y=log(x)+(x*x);
```

```
                        printf ("the result is  y=%f",y);
                    break;
                case 3:  y=pow(x,5)-2*pow(x,3)+12;
                         printf ("the result is  y=%f",y);
                   break;
   }
 return 0;
}
```

III. Контролни въпроси

1. Кои алгоритми са разклонени?
2. Какво е действието на конструкция ***if***?
3. Какво е действието на конструкция if-else?
4. Какво е действието на конструкция switch?
5. Какво е действието на конструкция break?

IV. Задачи за изпълнение

1. Да се създаде конзолно приложение за въвеждане на две числа от клавиатурата и намиране на по-голямото от тях. Задачата да се реализира в два варианта: чрез използване на конструкция if-else и, чрез оператор за условен израз.

2. Да се създаде конзолно приложение, чрез което се определя вида на триъгълник (равностранен, равнобедрен, разностранен) по зададени три страни.

3. Да се създаде конзолно приложение, което определя вид на триъгълник по за дадени 3 страни. Да се намери лицето на триъгълника като се използва Хиронова формула. В решението на задачата да е включена проверка за коректност на входните данни.

Упътване:

Ако a,b,c са страните на триъгълник, лицето на триъгълника се намира по следната формула: $s = \sqrt{p(p-a)(p-b)(p-c)}$ *като* $p = \frac{a+b+c}{2}$

Условието a, b, c да са страни на триъгълник е сборът на две съседни страни да е по-голям от третата: a+b>c, b+c>a, a+c>b

4. Да се състави вариант на алгоритъм и програма за пресмятане корените на квадратно уравнение.

5. Да се състави програма за намиране стойностите на функцията y по зададен аргумент x:

y=2, при x<=0;
y=x+2, при x (0,1);
y=3, при x [1,2];
y=5-x, при x (2,3);
y=2, при x>=3;

6. Да се състави алгоритъм и програма за пресмятане лице на: триъгълник, при k=1; квадрат, при k=2; правоъгълник, при k=3; кръг, при k=4. Стойностите за параметър k, както и данните за всяка фигура да се въвеждат от клавиатурата.

Лабораторно упражнение № 7
Итерационни конструкции. Реализация на циклични алгоритми

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е да се запознаят студентите с управляващите оператори ***while, do-while, for***; да се създадат умения за програмна реализация на задачи с циклични алгоритми.

II. Теоретична обосновка

Циклични алгоритми са тези при които има многократно повторение на група действия. Многократното повторение на група действия се нарича ***цикъл***. Конструкциите за цикли в С са част от управляващите, наричат се още **итерационни**. Предвидени са три итерационни конструкции, които реализират цикли в програмите: while, do-while, for. Тялото на цикъла се състои от един оператор. Ако алгоритъмът изисква в тялото да са повече от един оператори, те се обединяват чрез съставният оператор **{…}**.

1. Конструкция while

Чрез конструкция while се реализира цикъл с пред условие.Синтаксисът е следният:
while (условие) оператор;

Действието е следното: Проверява се условието. Ако стойността на израза е различна от 0 (TRUE), изпълнява се тялото на цикъла и отново се проверява условието за край, т.е.оператора в тялото за цикъла се изпълнява, докато условието е вярно. Когато стойността на условието стане равна на 0 (FALSE), излиза се от цикъла.

2. Конструкция за цикъл със след условие do-while

Конструкция do-while реализира цикъл със след условие. Синтаксисът е следния:
do оператор;
while (условие);

Действието е следното: Операторът в тялото на цикъла се изпълнява до тогава докато условието е вярно т.е изразът представящ условието има стойност различна от 0 (TRUE). Когато стойността на израза стане равна на 0 (FALSE), прекратява се изпълнението на цикъла.

3. Конструкция for

Итерационна конструкция for се използва предимно за реализация на цикли с известен брой повторения. По своето действие, конструкцията реализира цикъл с предусловие.

Синтаксисът е следния:
for (секция 1;секция 2;секция 3) оператор;

Секция 1 и секция 3 се състоят от един или няколко оператора, отделени със запетая. Секция 2 е условен израз.

Действието е следното: Изпълняват се операторите в секция 1. Това е т.нар. ***инициализираща секция***, която се изпълнява еднократно и с която се задават началните условия на цикъла. Изпълнението на for продължава в следната последователност: изчисляване на секция 2; изпълнение на оператора след скобите; изпълнение на секция 3. Тази последователност се изпълнява дотогава, докато стойността на израза в секция 2 има стойност различна от 0 (TRUE).

4. Примери за реализация на циклични алгоритми

4.1. Намиране на N факториел

```
// пример 1- намиране на N факториел чрез използване на оператор while
#include <iostream.h>

main()
{ int i;
  unsigned n,nf;

  cout<<" Input n=";
  cin>>n;

  nf=1;
  i=1;
  while(i<=n)
   { nf*=i;
     i++;
   }
 cout<<"\n N facturiel = "<<nf;
 return 0;
}

// пример 2: намиране на N факториел чрез използване на оператор do-while
#include <iostream.h>

main()
{ int i;
  unsigned n,nf;

  cout<<" Input n=";
  cin>>n;
  nf=1;
  i=1;
  do
   { nf*=i;
     i++;
   }  while(i<=n);
 cout<<"\n N facturiel = "<<nf;
 return 0;
}

// пример 3: намиране на N! чрез използване на оператор for

#include <iostream.h>

main()
{ int i;
  unsigned n,nf;

  cout<<" Input n=";
  cin>>n;

  nf=1;
  for(i=1;i<=n;i++) nf*=i;
  cout<<"\n N facturiel = "<<nf;
 return 0;
}
```

4.2. Намиране на средноаритметична стойност

Задачата е за намиране на средноаритметична стойност от произволен брой положителни числа, въведени от клавиатурата.

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

main ()
{ float x,sum,srst;
   int k;
   k=0;
   sum=0;
  do
    {  printf ("krai na vavejdaneto e 0\n");
        printf ("vavedi  x na broi polojitelni chisla:\n");
         scanf ("%f",&x);

        if (x>0)
          { k++;
             sum+=x;
           }
   }  while(x!=0);
  srst=sum/k;
  printf ("srst=%f",srst);
  return 0;

}
```

III. Контролни въпроси

1. Кои алгоритми са циклични?
2. Какво е действието на оператор ***while***?
3. Какво е действието на оператор ***do-while***?
4. Какво е действието на оператор ***for***?
5. Пояснете разликите между оператори ***while*** и ***do-while***?

IV. Задачи за изпълнение

1. Да се създаде конзолно приложение, което намира сумата на произволно въведени положителни числа от клавиатурата. Въвеждането на стойност 0 да прекрати понататъшното въвеждане на числа.
2. Да модифицира програмния код в задача 1, така че приложението да намира произведението на произволно въведени положителни числа от клавиатурата.
3. Да се създаде конзолно приложение, с което се извеждат на екрана стойностите от 1 до n в прав и обратен ред, като стойността на n се въвежда от клавиатурата.
4. Да се създаде конзолно приложение, което отпечатва на екрана следното:

```
1
1 2
1 2 3
1 2 3 4
…
1 2 3 4 … n
```

Стойността на n се въвежда от клавиатурата.

5. Да се създаде конзолно приложение, което отпечатва на екрана следното:

```
       0
      101
     21012
    3210123
n..3210123..n
```

Стойността на n се въвежда от клавиатурата.

6. Да се състави алгоритъм и програма за намиране на e^x в ред на Фурие. Натрупването на междинната сума да се прекрати, когато поредния член стане по-малък от 10^{-8} . Формулата за пресмятане е: $e^x=1+x/1+x^2/2!+x^3/3!+...+x^n/n!$

Да се сравни получената стойност със стойността, получена чрез използване на функция exp().

Лабораторно упражнение № 8
Дефиниране и обработка на масиви

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е да се запознаят студентите с масивите, като структурирани типове данни; да се създадат умения за програмна реализация на задачи с масиви.

II. Теоретична обосновка

11. Дефиниране и използване на масиви

Типовете данни в програмните езици се разделят най-общо на ***скаларни*** и ***структурирани***. Скаларни типове данни са тези, които се състоят от един елемент. В C, пример за такива данни са аритметичните типове ***int, char, float*** и ***double***. Структурираните типове се състоят от повече от един елемент

Масивът е структура от данни, състояща се от множество последователно наредени елементи от един и същи тип, достъпът до който се осъществява чрез името на масива поредния номер на елемента т.нар. **индекс**.

Едномерни масиви се дефинират чрез следната декларация:
име_тип име_масив[размерност];

име_тип - определя типа на елементите на масива. Може да е някой от стандартните типове или дефиниран от програмиста;

име_масив е идентификатор, определящ името на променливата от тип масив;

размерност - константен израз, който определя броя на елементите в масива и се нарича още ***граница***.

Примери:

```
int massiv[10];        // дефиниране на едномерен масив massiv от 10 цели елемента
char ch[20];           // дефиниране на едномерен масив ch от 20 символа
float a[10], b[20];    // дефиниране на два едномерни, реални масива a и b от 10 и 20
                       // елемента, съответно
```

Достъпът до даден елемент на масива е с името на масива и индекса на елемента.

Пример:

```
a[2]=0;
b[18]=1;
ch[10]='y';
```

Особеност на език C, относно масивите е, че индексите на елементите започват от 0, т.е. ако даден масив е с N елемента, първият елемент е с номер 0, а последният е с номер N-1.

Дефинирането на многомерни масиви е чрез следната декларация:
име_тип име_масив[граница1][граница2].....;

Примери:

```
float ax[10][20];      // дефиниране на двумерен масив ax от 10x20 реални елемента
int bx [10][10][5];    // дефиниране на тримерен масив bx от 10x10x5 целочислени
                       // елемента
```

При дефинирането на масиви могат да бъдат зададени начални стойности на елементите, т.е. масивите да бъдат инициализирани. Общият вид на дефиницията е следния:

име_тип име_масив [размерност]={списък стойности};

Стойностите на елементите се разделят със запетаи.
Пример:
int array[3] = {1, 3 , 5};
/* съответно стойностите на елементите са: array[0]=1; array[1]=3; array[2]=5 */

2. Примери за използване на масиви

В упражнението е представено решението на две примерни задачи, свързани с използването на масиви като структури от данни.

2.1. Търсене на минимална стойност

Задачата е да се намери позицията и стойността на минимален елемент в едномерен масив от 10 целочислени елемента.

```
#include <stdio.h>

main ()
{ int mas[10];
  int i,j,a,k, min;

 printf ("insert the number of the massive=");
 scanf ("%d",&j);

  for(a=0;a<j;a++) scanf("%d",&mas[a]);

  min=mas[0];                    // min съдържа стойността на най-малкият елемент
  k=0;                          // k съдържа позицията на най-малкият елемент
  for (i=1; i<j;i++)
        if (mas[i]<min)
              { min=mas[i];
                k=i;
              }
  printf ("min=%d\n",min);
  printf ("poziciqta e %d\n",k);
  return 0;
}
```

2.2. Сумиране на матрици

В примерната програма е решението на задачата за въвеждане на стойностите на две матрици (a, b)и намиране на тяхната сума в резултантна матрица c.

```
#include <iostream.h>

void main()

{ const M=20;  // максималният размер на матриците е 20x10
  const N=10;
  int a[M][N], b[M][N], c[M][N];
```

```
  int i,j,m,n;      // m,n – действителен размер на матриците

  do {
         printf("m=");
        scnaf(“%d”,&m);
      } while ((m>20)||(m<2));          // стойността на m да е в интервала [3,20]
  do {
         printf("n=");
        scnaf(“%d”,&n);

      } while ((n>10)||(n<2));           // стойността на n да е в интервала [3,10]

  for(int i=0;i<m;i++)
   for(int j=0;j<n;j++)   scanf(“%d”,&a[i][j]);       // въвеждане на матрица a

  for(int i=0;i<m;i++)
    for(int j=0;j<n;j++)   scanf(“%d”,&b[i][j]);      // въвеждане на матрица b

  for(int i=0;i<3;i++)                               // сумиране на матриците
   for(int j=0;j<3;j++)   c[i][j]=a[i][j]+b[i][j];

  for(int i=0;i<3;i++)                               // извеждане на матрица c
     {for(int j=0;j<3;j++) printf(“%d   “,c[i][j]);
      printf(“\n”);;
      }
}
```

IV. Контролни въпроси

1. Какво представлява структурата данни ***масив***?
2. Дефинирайте едномерен масив image от 20 целочислени елементи?
3. Дефинирайте двумерен масив matrix от 10x20 реални елементи?
4. Дефинирайте символен низ string с максимален размер 20 символа?
5. Как се инициализират масиви при тяхното дефиниране

III. Задачи за изпълнение

1. Да се създаде конзолно приложение, с което се въвеждат 10 цели числа и се извеждат в прав и обратен ред.

2. Да се създаде конзолно приложение, с което се намира сумата и средно аритметично на елементите на едномерен масив от 10 реални числа с двойна точност.

3. Да се създаде конзолно приложение, с което се намира стойността и позицията на максималния елемент в едномерен масив от 10 целочислени елемента.

4. Да се създаде конзолно приложение транспониране на матрица, представена като двумерен масив от 4x3 реални елемента. Стойностите на елементите на матрицата да се въвеждат от клавиатурата.

5. Да се създаде конзолно приложение за въвеждане елементи на двумерен масив и намиране сумата на елементите във всеки ред.

6. Да се създаде конзолно приложение, чрез което се въвежда двумерен масив от 5x5 целочислени елемента и се нулират (задава се стойност 0) всички елементи, разположени по двата диагонала.

Лабораторно упражнение № 9
Дефиниране и обработка на символни низове

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е да се запознаят студентите със символните низове; да се създадат умения за програмна реализация на задачи с масиви от символи.

II. Теоретична обосновка

За разлика от други програмни езици, в език С не е предвидена възможност за дефиниране на променливи от тип ***низ*** (***string***). За целта, като низове се използват масиви от символи т.е. масиви от тип ***char.***

Когато се дефинира символен низ, трябва да се предвиди следната особеност: ако даден низ съдържа **n** символа, размерът на масива от символи е **n+1**. Последен символ е *управляваща константа за край на низ*: ***'\0'*** (NULL), който автоматично се добавят при въвеждане на низа чрез ***cin*** или ***scanf().*** Пример:

```
char str[20];
```

В случая, дефиниран е символен низ, който може да съдържа 19 символа максимално, последният символ в низа е управляващата константа за край на низ.

Пример за програма, която работи с масиви от символи е следната задача: Да се намери колко пъти се съдържа даден символ в низ.

```
#include <stdio.h>

main()
{
      char str[256];
      char ch;

      printf("Enter a character:");
      scanf("%c",&ch);
      printf("Enter a string:");
      scanf("%s",str);

      int counter = 0, i = 0;
      while(str[i]!='\0')
      {
            if(str[i]==ch) counter++;
            i++;
      }
      printf("\nCharacter %c contains %d times
in string %s",ch,counter,str);
      return 0;
}
```

Масив от символи може да бъде инициализиран по два начина:

- При дефинирането на масив от символи се присвоява като стойност низова константа (поредица от символи, затворени в апострофи). В случая, добавянето на управляваща константа '\0' за карай на низ е автоматично. Пример:

```
char str[]= ”abcd”;
```

Масивът str съдържа 5 елемента; последният е със стойност '\0'.На всеки елемент се присвоява определена стойност. В този случай константата '\0' се присвоява автоматично на последният елемент на масива.

- Друг начин за инициализиране на символен низ е този, по който обикновено се задават начални стойности на елементите на масива:

```
char str[]= {'a','b','c','d','\0'};
```

В този случай, на последният елемент задължително се задава стойност '\0'. Разбира се, вторият начин е твърде неудобен и затова не се практикува.

За работа с низове в стандарта ANSI C са предвидени различни функции, чиито декларации са в заглавен файл ***string.h***. Най-често използваните функции за работа с низове са:

- strcpy()

Функцията има следния синтаксис:

char *strcpy(char *to, const char *from);

Функцията копира символите от низ from в низ to, включително и символът за край на низ. Трябва да се отбележи, че функцията не прави проверка дали двата низа не са един и същ и е възможно препокриване на двата низа.

- strncpy()

Функцията има следния синтаксис:

char *strncpy(char *to, const char *from, int count);

Функцията копира най-много count на брой символа от низ from в низ to, като добавя и символът за край на низ. Функцията връща резултантния низ.

- strlen()

Функцията има следния синтаксис:

int strlen(char *str);

Функцията намира дължината на низа и връща като стойност броя на символите в низа, без символът за край на низ.

- strcmp()

Функцията има следния синтаксис:

int strcmp(const char *str1, const char *str2);

Функцията сравнява два низа и връща следните стойности:

по-малка от 0 (нула), ако str1 е по-малък от str2;

0 (нула) – ако низовете са еднакви;

по-голяма от 0 (нула), ако str1 е по-голям от str2;

Трябва да се отбележи, че използването на тези функции изисква включване на заглавен файл ***string.h***:

#include <string.h>

III. Задачи за изпълнение

1. Да се създаде конзолно приложение за въвеждане на символен низ и проверка колко пъти се среща определен символ в низа.

2. Да се създаде конзолно приложение, с което се въвежда символен низ и всеки срещнат интервал се заменя със знак за табулация

Лабораторно упражнение № 10
Указатели. Адресна аритметика

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е запознаване на студентите с работа с указатели, предимствата на използването на указатели, както и с принципите на адресната аритметика; да се създадат умения за използване на указателите в програмите.

II. Теоретична обосновка

Всеки елемент от програма (константа, променлива) се съхранява в определени клетки от паметта или, както е прието да се казва: на определен адрес от паметта.

Променлива, чиято стойност е адрес от паметта се нарича ***указател***. Стойността на указателя посочва (указва) местоположението на дадена променлива и косвено служи за достъп до нея, за разлика от познатия, директен достъп до променливата, чрез името й.

Като обобщение на казаното, ***указателят е променлива, чиято стойност е адрес на променлива.***

Използването на указатели е начин за писане на ефективни и компактни програми. От друга страна, ако не бъдат правилно използвани, указателите могат да доведат до разрушаване на данните.

Както всяка променлива и указателят трябва да се дефинира. Общият вид на дефиницията на променлива-указател е:

тип *име_указател [=стойност];

Тип определя типа на съдържанието на указателя т.е. типа на променливата, чийто адрес ще сочи указателя.

Име_указател е идентификатора на променливата указател. Символ * пред името на променливата определя, че променливата е указател. Обикновено, имената на променливите указатели започват с **p**, от английският термин **pointer** (указател).

Както всяка друга променлива, така и указателят може да бъде инициализиран с дадена начална стойност. Тази стойност трябва да бъде адрес на променлива.

Примери:

```
int *px;        // px е указател към променлива от тип int
float *p1;      // p1 е указател към променлива от тип float
double *p2;     // p2 е указател към променлива от тип double

int x;
int *px=&x;     // указателят px сочи адреса на променливата x
```

Ако указателят не е инициализиран, неговата стойност е NULL. Тази стойност може да бъде присвоена на указател от всеки тип.

2. Операции с указатели. Адресна аритметика

Съществуват две специални операции с указатели, които позволяват тяхното ефективно използване:

операция **&** определя адреса на операнда

операция ***** определя стойността от посочения адрес.

Форматът на операциите & и * е следния:

&променлива

Освен променлива, операндът може да е и елемент на масив. Като резултат операцията връща адреса на променливата.

указател
Резултат от операцията е стойността на променливата, чийто адрес съдържа указателя.

Върху указателите могат да се прилагат по-малко на брой аритметични операции, отколкото върху обикновените променливи. Освен това, изпълнението на аритметичните операции върху указатели е свързано с някои особености, поради което тези операции са известни още като ***адресна аритметика.***

Особеност на адресната аритметика е, че при изпълнението на операциите *събиране, изваждане, инкрементиране и декрементиране* автоматично се извършва ***мащабиране,*** което отчита типа на обектите, към които сочат указателите.

4. Указатели и масиви

В C/C++ съществува пряка връзка между указатели и масиви - ***имената на масивите се явяват указатели***, като съдържат адреса на първия елемент от масива т.е. елемент с индекс 0.

По този начин, достъпът до елемент на масив може да се осъществи и чрез указател.

Пример:

```
int array[5];
int *p;
int x,y,i;

x=array[0];      // променливите x, y осъществяват достъп до един и същ елемент
y=*array;

x=array[i];      // променливите x, y осъществяват достъп до елемент с индекс i
y=*(array+i);

p=array+i;       // също, достъп до елемент с индекс i
y=*p
```

Основната разлика между името на масива и променливата-указател е, че името на масива се явява константа и стойността му не може да бъде променена. Например:

```
p=array;
p++;             // изразите са допустими

array=p;
array++;
array=&x;        // изразите са недопустими
```

Достъпът до елементите на масив може да се осъществи както чрез индекс, така и чрез указател. Достъпът чрез указател е много по-бърз, отколкото този чрез индекс и поради тази причина често е предпочитан в програмите на C/C++.

Пример за достъп до елементите на масив чрез указател е даден с програмната реализация на задачата за сумиране на елементите на едномерен масив mas от 10 целочислени елемента. Указателят p сочи текущият елемент от масива.

```
// намиране ма сумата от елементите на едномерен масив
#include <stdio.h>
main()
{ int mas[10],*p;
  int i,sum;
```

```
    p=mas;
   for(i=0;i<10;i++)
    { printf ("insert the elements of the masive=");
       scanf ("%d",p);            // достъпът де текущия елемент е чрез указател
       p++;
    }
   p=mas;
   sum=0;
     for (i=0;i<10;i++)
       { sum+=*p; // достъпът де текущия елемент е чрез указател
         p++;
        }
   printf ("sum=%d",sum);
   return 0;
}
```

5. Указатели към указатели

Тъй като указателят също е променлива, която се разполага на определен адрес, възможно е, друг указател да сочи неговият адрес. такъв тип указател се нарича ***двоен указател***.

Дефинирането на двоен указател се извършва по следният начин:

тип **име[=стойност];

Например, нека x е променлива; px - указател към променливата и нека ppx да е указател, съдържащ адреса на px. Дефинирането на тези променливи е по следният начин:

```
int x;
int *px=&x;
int **ppx=&px;
```

Приложението на двойните указатели е при работата със различни структури от данни като дървета, списъци, графи, както и при работа с многомерни масиви.

III. Контролни въпроси

1. Какво се дефинира със следния програмен ред?

```
int* px, py, pz;
```

2. Има ли разлика ако дефиницията е записана `int *px, py, pz;` ?

3. Напишете програмен ред, с който дефинирате 3 указателя към тип int.

4. Какво ще се изведе на екрана след изпълнението на програмен код:

```
int* px, py;
py=10;
px=&py;
++*px;
printf("%d\n", py);
```

5. Какво ще се изведе на екрана, ако програмен ред 4 се замени с програмен ред: `*++px;`

IV. Задачи за изпълнение

1. Да се за размяна на две числа, като достъпът се осъществява чрез указатели.

2. Да се създаде конзолно приложение, с което се дефинира едномерен масив от 10 целочислени елемента. Да се въведат стойностите на елементите и да се изведат като се използват три начина: достъп чрез индекс посредством оператор [], достъп чрез индекс посредством адресен оператор *, достъп чрез указател

3. Да се състави конзолно приложение за търсене на най-голям и най-малък елемент в едномерен масив от 10 реални елемента. Достъпът до елементите на масива да се осъществи чрез указатели.

Лабораторно упражнение № 11
Програмни модули в C/C++. Функции

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение да бъдат запознати студентите с функциите, като програмните модули в C/C++. Обръща се внимание, че дадена програмна задача може да се раздели на отделни подзадачи. Целта на упражнението е да се създадат навици и умения за реализацията на функциите, които решават отделните подзадачи, както и тяхното използване в една обща програма.

II. Теоретична обосновка

1. Видове програмни модули

В програмните езици, за видовете програмни модули се използва следната класификация:

- главна програма;
- подпрограми (процедури, функции).

Главна програма е програмен модул, на който операционната система предава управлението при стартиране на изпълнимия код. При изпълнението си, главната програма може да предаде управлението на други програмни модули, наречени ***подпрограми***. След завършване, главната програма връща управлението на операционната система.

Подпрограма е програмен модул, който получава управлението от друг програмен модул т.е. подпрограмата се извиква от друг модул. След приключване, подпрограмата връща управлението в програмния модул, от където е извикана.

За разлика от други програмни езици, където подпрограмите се разделят на процедури и функции, в C и C++ съществува само един тип програмни модули - ***функции***.

Функция е самостоятелен фрагмент от програма, съдържащ описания на променливи и набор оператори на езика. Фрагментът е затворен във скоби {…} и в крайна сметка се изпълнява като един обобщен оператор.

Всяка програма на C или C++ се състои от една или повече функции. Задължително трябва да съществува една главна функция с име ***main()***. Функция ***main()*** изпълнява роля на главна програма. Това е функцията, към която първоначално се предава управлението от операционната система. След приключване на изпълнението, функция ***main()*** връща управлението на операционната система.

Като самостоятелен програмен модул, функцията може да бъде извиквана многократното, като се стартира с различни входни данни (променливи). Входните променливи се наричат още ***параметри*** на функцията.

След завършването си функцията връща ***резултат***. Като програмен модул функцията връща чрез името си, един резултат.

Параметрите и ***резултатът*** са връзката на функцията с останалите програмни модули. Идея за използването на функции е те да могат да се извикват с различни входни данни.

2. Дефиниране на функции. Оператор return

Дефиниция на функция представлява цялостното описание на функцията. Описанието на функцията се състои две части:

- **заглавна част (*прототип*);**

- **тяло,** което съдържа декларации на локални променливи и оператори.

Общият вид на дефиницията на функция е следният:

```
тип  име (списък формални параметри)        // прототип на функция
{                                            // тяло на функция
   // описание на локални променливи
   // оператори;
}
```

Тип на функцията е типът на стойността, която функцията ще върне като резултат. По подразбиране функциите са от тип ***int*** т.е. ако не бъде записан тип, счита се, че функцията е от тип ***int***. Чрез името си, функцията връща един резултат.

Име на функцията е идентификатор, чрез който еднозначно се определя функцията.

Формални параметри са списък входни променливи, чрез който се осъществява връзката между функцията и останалите функции. Те имат описателно значение.

Списък формални параметри се представя в следния вид:
(тип1 променлива1 , тип2 променлива2, ...)

Тялото на функцията включва множество декларации на локални променливи и оператори, реализиращи задачата на функцията.

3. Извикване на функция. Предаване на параметри

Извикването на функция става чрез името и. При извикване на функцията, в скобите след името се задават ***фактическите параметри*.** Извикването на функция се задава като:

име (списък фактически параметри);

Подобно на формалните, фактическите параметри са разделени със запетаи.

Фактическите или **действителни** параметри са константи, променливи или изрази, които при извикване на функцията предават своите стойности на формалните параметри, за да се изпълни функцията. Този процес се нарича ***свързване на формалните с фактическите параметри*.**

Фактическите и формалните параметри си съответстват по брой и по тип.

4. Декларации на функции

В случай, че се наложи функцията да бъде извикана преди нейната дефиниция, необходимо е, функцията да бъде декларирана.

Декларацията на функцията е нейният прототип (заглавната част на функцията) последван от символ ***;***

5. Примери за реализация на функции

5.1. Намиране на N факториел

// намиране на N факториел чрез използване на функция

```
#include <stdio.h>
int Factoriel(int n);


main()
{
      int n,fact;
      printf("n=");
```

```
	scanf("%d",&n);
	fact = Factoriel(n);
	printf("N Factoriel = %d",fact);
	return 0;
}

int Factoriel(int n)
{
	int nf=1,i;
	for(i=1;i<=n;i++)
		nf*=i;
	return nf;

}
```

III. Контролни въпроси

1. Как се дефинира функция?
2. Какво е прототип на функция?
3. Защо е необходимо да се декларира функция?
4. Какви са разликите между фактически и формални параметри на функция?
5. Какво представлява процеса свързване на фактическите с формалните параметри?

IV. Задачи за изпълнение

1. Да се създаде конзолно приложение, което пресмята лицата на различни фигури: кръг, правоъгълник, триъгълник. Пресмятането на лицата на фигурите да се извършва с отделни функции. В проекта да се включи и функция, която проверява дали стойностите за страните на триъгълника са валидни.

2. Да се дефинира функция, която намира разстояние между две точки, координатите на който се предават като параметри. Да се създаде конзолно приложение, което намира разстояние между обща дължина на начупена линия.

Упътване: Координатите на точките да се дефинират като два отделни масива – първият да съдържа x- координатите, а втория – y-координатите.

3. Да се състави конзолно приложение за намиране на e^x в ред на Фурие. Натрупването на междинната сума да се прекрати, когато поредния член стане по-малък от 10^{-8} . Формулата за пресмятане е: $e^x=1+x/1+x^2/2!+x^3/3!+...+x^n/n!$

Да се сравни получената стойност със стойността, получена чрез използване на функция exp().

Упътване: Да се използва функция за намиране стойността на n!.

Лабораторно упражнение № 12
Предаване на параметри чрез указатели. Рекурсивни функции

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е да се разширят знанията на студентите за използване на функциите като програмни модули; да се разгледа механизма на предаване на параметрите на функциите чрез указатели, областта на действие на променливите и да се дадат знания за използване на рекурсивни функции.

II. Теоретична обосновка

1. Предаване на параметри чрез указатели

Функциите взаимодействат помежду си чрез: *предаване на параметри, връщане на стойност* и чрез *външните (глобални)* променливи.

Предаването на параметри е по два начина:

- ***по стойност;***
- ***чрез указател.***

Особеност на предаването на параметри чрез стойност е, *че фактическите параметри, с които се извиква функцията, се копират в стека и функцията не работи с оригиналните параметри, а с техни копия*. По този начин, *функцията не може да променя стойностите на оригиналните параметри*.

Когато е необходимо функциите да работят с оригиналните параметри, а не с техните копия, параметрите трябва да бъдат предадени чрез указатели.

- предаването на параметрите на функциите чрез указатели се регламентира по следните правила:
- формалните параметри на функцията се декларират като указатели;
- в тялото на функцията се използва съдържанието на указателите, като формални параметри;
- при извикване на функцията, фактическите параметри също трябва да са указатели.

Пример за предаване на параметри чрез указатели е функцията за размяна на две числа swap():

```
#include <iostream.h>

void swap(int *, int *);
main()
{ int p,q;
  cin>>p;
  cin>>q;
  swap(&p,&q);                  // на x и y се предават адресите на p и q
  cout<<"p="<<p<<"q="<<q;
}
void swap( int *x, int *y);      // x,y са указатели
{ int b;
  b=*x;                                 // разменя се съдържанието на клетките на адреси x,y
  *x=*y;
  *y=b;
   return;
```

}

При предаването на параметрите чрез указатели трябва да се има предвид, че функцията работи с оригиналните стойности на параметрите, а не с техни копия и може да промени оригиналните стойности. Неправилното използване на предадените чрез указатели параметри, може да доведе до разрушаване на необходима информация.

Предаването на параметри чрез указатели се използва предимно, когато е необходимо функцията да върне повече от един резултат. Тъй като стойностите на тези параметри могат да бъдат променяни, те се използват като ***входно-изходни*** параметри.

2. Рекурсия

В C/C++ функциите са ***рекурсивни***. ***Рекурсия*** *е механизъм, при който дадена програма може да има обръщение сама себе си.*

Има два варианта на рекурсия: ***пряка*** и ***косвена***. При пряката рекурсия в тялото на дадена функция има обръщение към самата нея, т.е. функцията се извиква сама себе си.

Косвена рекурсия се определя, когато дадена функция извиква втора функция, която от своя страна прави обръщение към първата. Косвената рекурсия може да е на повече нива. Например, първа функция извиква втора функция; втора функция извиква трета функция; третата функция извиква първата.

Рекурсивните функции са особено подходящи, когато трябва да се реализират алгоритми, които са рекурсивни по определение.

Примери за такива алгоритми са: изчисляване на факториел, обработка на дървовидни структури от данни и др.

При използване на рекурсивни функции, винаги трябва да се предвиди ***условие за изход от рекурсията*** т.е. функцията да се извиква (или да не се извиква) при определено условие. Ако не бъде предвидено такова условие, рекурсивното обръщение към функцията ще се превърне в ***безкраен цикъл***.

Доказано е че всяко рекурсивна функция има интерактивен вариант т.е. всеки рекурсивен алгоритъм може да се реализира без използване на рекурсия.

Пример за рекурсивна функция е даден чрез програма за намиране на N факториел.

```
// намиране на N факториел чрез използване на функция

#include <iostream.h>

int facturiel(int );       // декларация на функцията
main()
{   unsigned n,f;

  cout<<" Input n=";
  cin>>n;
       f=facturiel(n);                                                    //
                              извикване на функция facturiel с параметър n
  cout<<"\n N facturiel = "<<f;
 return 0;                              // код за връщане на функция main в ОС
}
unsigned facturiel(unsigned n)          // дефиниране на функцията facturiel
{ int nf;
  nf=1;
  if (n<=1) nf=1;                       // връщане от рекурсията
  else nf=n*facturiel(n-1);             // рекурсивно извикване на функцията
return nf;                              // връщане на резултат
```

}

При използване на рекурсивни функции трябва да се има предвид, че автоматичните променливи, които се създават при всяко извикване на дадена функция заемат съответното място в стека. При голяма дълбочина на рекурсията трябва да се предвиди достатъчно голям по обем стек.

III. Задачи за изпълнение

1. Да се тества представената програма за размяна на две числа.

2. Да се състави функция за намиране стойността и позицията на максималния елемент от едномерен масив от N елемента ($N \in [5,20]$).

3. Да се състави програма за въвеждане на матрица от 10x20 целочислени елемента. Да се намерят стойността и позицията на минималния елемент във всеки ред. Данните да се запишат в двумерен масив b[10][2], като първият стълб съдържа позицията, а втория – стойността на минималния елемент. Търсенето на минималния елемент да е чрез функция.

4. Да се тества рекурсивната функция за пресмятане на N факториел.

IV. Контролни въпроси

1. Кога се налага предаване на параметрите чрез указатели?
2. Кои функции са рекурсивни?
3. Каква е разликата между пряка и косвена рекурсия?
4. Какви са характеристиките на външните променливи?
5. Какви са характеристиките на автоматичните променливи?

Лабораторно упражнение № 13
Сложни типове данни от динамичен тип. Структури, обединения

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е запознаване на студентите със сложни типове данни в C/C++ структури, обединения и разредни полета; създаване на умения за организиране на данните и приложение на структурите и обединенията в програмите.

II. Теоретична обосновка

Типовете данни в програмните езици могат да бъдат класифицирани като:

- ***прости*** - типове данни, които се състоят от един елемент (компонент);
- ***структурни*** - типове данни, които се състоят от повече от един елемент.

Типичен пример за данни от структуриран (сложен) тип са масивите.

В език С като структуриран тип данни могат да бъдат класифицирани: структурите (struct) и обединенията (union). Тези типове данни могат да се използват за реализация на списъци и дървовидни структури.

1. Структури (*struct*)

Структурата е сложен тип данни, състоящ се от повече от един разклонени елемента. Структурата има еквивалентен тип ***Record (запис)*** в програмен език Pascal.

1.1. Деклариране на структура

Чрез декларирането на структура се създава нов тип данни, който е дефиниран от програмиста. Декларацията на структурата има следният общ вид:

```
struct   име
  { тип  име1;   //описание на първи елемент
    тип  име2;   //описание на втори елемент
     ...
   };
```

Структурата представлява поредица от елементи, които в общия случай са от различен тип.

Име е валиден идентификатор, служещ за дефиниране на структурата. Името се предшества от ключова дума ***struct.***

В блока се описват елементите на структурата, определени чрез техните ***имена*** (***идентификатори***) и техният ***тип***.

Пример:

```
struct dot                          // структура dot , описваща точка
 { int x;                           // с координатите: x,y и цвят:color
   int y ;
   unsigned char color;
 }
```

Чрез декларацията на структура не се дефинира променлива, а се определя нов тип т.е. явява се описание на шаблон за този тип. След като бъде декларирана дадена структура, в последствие, тя може да се използва като всеки един от типовете за дефиниране на променливи от този тип.

1.2. Операции със структури

Структурите могат да участват като операнди в следните оператори:

- прилагане на адресна операция **&**;
- обръщение към елемент от структура (операции **.** и **->**);
- инициализиране на структури.

Адресен оператор извличане на адрес (**&**) може да се приложи към променлива – структура, така както при обикновените променливи.

Пример:

```
struct dot
 { int x;
   int y;
} current, *pcurrent;

pcurrent = &current;   //променлива pcurrent сочи ( съдържа адреса на )
                       // структура current
```

Обръщането към елемент от структура може да се извърши чрез една от следните оператори :

- оператор **.** (***извличане на елемент от структура, чрез променлива структура***) се прилага върху променлива-структура в следния вид:

променлива_структура .елемент

Пример :

```
current.x = 10;
current.y = 20;
```

- оператор **->** (***извличане на елемент от структура чрез указател***) се прилага към указател от тип структура в следния вид:

указател към структура -> елемент

Пример:

```
pcurrent=&current;
pcurrent->x=10; //задава стойности 10,10 за координатите x и y на текущата
pcurrent->y=10; // точка
```

Променлива структура се инициализира като всяка променлива - след знак за присвояване във фигурни скоби се записват поредица от начални стойности за всеки елемент от структурата.

Пример :

```
struct dot current  = {100, 28};
```

1.3. Съвместно използване на структури и масиви

Масивите и структурите могат да се използват съвместно. Възможно е да се дефинира масив от структури или масив да е елемент на структура.

В случай, че се дефинира масив от структури, дефиницията има следния общ вид:

struct име_структура име масив [граница];

2. Обединения (union)

В език С е предвидена възможност да се дефинират области от паметта, в които по различно време да се записват променливи от различен тип. Такъв тип описания се наричат ***обединения (union)***.

Декларацията на обединение е подобна на тази на структура и има следния синтаксис:

```
union име
{ тип  име1;   //описание на първи елемент
  тип  име2;   //описание на втори елемент
   ...
};
```

Декларацията на обединение започва с ключова дума ***union***. ***Име*** е валиден идентификатор, служещ за дефиниране на обединение.

В блока се описват елементите на обединението, определени чрез техните ***имена*** (***идентификатори***) и техният ***тип***.

Както и при структурите, чрез декларацията на обединение се създава нов тип данни, а не се заделя памет за съответна променлива.

Пример:

```
union set
 { int k;
   char c;
   float f;
};
```

Основната разлика между структура и обединение е, че при дефиницията на променлива от тип структура (***struct***), компилаторът заделя необходимата памет за всички елементи, включително и необходимите изравнявания, докато при дефиницията на променлива от тип обединение (***union***) компилаторът заделя памет, достатъчна за записа на най-големия елемент.

Достъпът до елементите на обединение е чрез същите операции, както и достъпа до елементите на структура:

- операция **.** (***извличане на елемент от структура, чрез променлива структура***);
- операция -> (***извличане на елемент от структура чрез указател***).

3. Разредни полета

В език С е възможно да се дефинират и използват набори от последователни битове. Такива набори се наричат ***разредни полета (bit fields)***. Чрез разредните полета се осъществява връзката между високото ниво на езика и ниското ниво на апаратната част.

Разредно поле се декларират като елемент на структура с използване на следния синтаксис:

unsigned int име_поле:размер;

Име_поле е идентификатор, определящ елемента;

Размер е броят битове за полето. Размерът е цяло число в интервала [0,16] (за някои компилатори интервалът е [0,15], заради знаковия бит).

Разредните полета са машинно зависима езикова конструкция. Правилата за тяхното дефиниране до голяма степен зависят от конкретния компилатор.

Достъпът до елемента разрядно поле се осъществява както достъпа до елемент на структура чрез операции:

- операция **.** (***извличане на елемент от структура, чрез променлива структура***);
- операция -> (***извличане на елемент от структура чрез указател***).

III. Контролни въпроси

1. Какво представлява типа данни ***структура***?
2. Какво представлява типа данни ***обединение***?
3. Къде намират приложение ***разредните полета***?
4. Какво е действието на операция **.** (точка)?
5. Какво е действието на операция -> ?
6. Как се осъществява достъп до елемент на масив от структури?

IV. Задачи за изпълнение

1. Да се декларира структура, описваща точка в равнината. Да се създаде конзолно приложение, с което се дефинират две точки, въвеждат се стойности за координатите и се извеждат. Да се допълни функционалността на приложението като се въведе масив от точки, въведат се и се изведат стойностите на координатите.

2. Да се декларира структура, описваща правоъгълник. Да се дефинира функция, която намира лице на правоъгълник. Да се създаде приложение, с което се дефинира масив от правоъгълници, въвеждат се стойности за страните и се намират лицата на фигурите.

3. Да се декларира структура, описваща комплексно число. Да се дефинират функции, които намират сбора и разликата на две комплексни числа. Да се създаде демонстрационна програма, която симулира работата на калкулатор за комплексни числа

Лабораторно упражнение № 14
Работа с файлове при буфериран достъп

I. Цел на лабораторното упражнение

Целта на лабораторното упражнение е запознаване с функциите за работа с файлове в C/C++; създаване на умения за използване в тези функции в различните потребителски програми.

II. Теоретична обосновка

1. Работа с файлове в C/C++

При обработката на данни чрез програмните езици е необходимо данните да бъдат четени от файл или съхранявани във файл. Във всеки програмен език се предвижда възможност за работа с файлове – достъп до файл, четене на данни от файл, запис на данни във файл.

Съществуват два основни начина за достъп до дискови файлове:

- буфериран;
- небуфериран.

При буферирания достъп, от файла се чете или записва символ по символ, като програмистът не се интересува от системно зависими особености като размери на буфери, сектори и др. Този тип достъп се нарича още ***вход-изход от високо ниво***. При него, функциите автоматично осигуряват необходимите буфери.

2. Функции за работа с файлове при буфериран достъп

Въпреки различията в отделните компилатори, обикновено те поддържат едни и същи стандартни функции за работа с файлове т.е. имената на функциите, типа, както и изискваните параметри са едни и същи. Функциите за работа с файлове при буфериран достъп са дефинирани в стандартния заглавен файл ***stdio.h***

2.1. Отваряне на файл

Създаването на нов файл или отварянето на съществуващ файл за четене или запис при буфериран достъп, се осъществява чрез функция ***fopen()***.

Функцията има следния прототип:

FILE *fopen(char *pathname, char *type);

Функцията ***fopen()*** връща като резултат указател към тип FILE. Структурата FILE е дефинирана във файла ***stdio.h*** и в нея се записват всички необходими данни за файла. Ако има грешка при отваряне на файла, функцията връща резултат NULL.

Параметрите на функцията ***fopen()*** са:

char *pathname – указател към низ, съдържащ името на файла; името на файла трябва да е валидно за съответната операционна система;

char *type – указател към низ, определящ допустимите операции във файла.

2.2. Затваряне на файл

Затварянето на нов при буфериран достъп, се осъществява чрез функция ***fclose()***.

Функцията има следния прототип:

int fclose(FILE *fp);

Като параметър, функцията ***fclose()*** изисква указател към тип FILE. Функцията затваря файл, отворен чрез ***fopen()***. По този начин се освобождават всички системни ресурси, заети от отворения файл. Освен това, функцията съхранява на диска незаписаните данни.

2.3. Запис на данни във файл

Примери за функции, които осъществяват запис във файл при буфериран достъп са:
fputc() – записва символ в поток;
fputs() – записва низ в поток;
fprintf() – записва форматирани данни в поток;
fwrite() – записва блок данни в поток.

Функцията ***fputc()*** има следния прототип:
int fputc(int c, FILE *stream);

Функцията записва символ, определен от първия параметър (***int c***) в поток, определен от втория параметър (***FILE *stream***). Потокът е отворен файл за запис. Функцията връща или записания символ, при нормално приключване, или символ EOF, ако е настъпила грешка.
Функцията ***fwrite()*** има следния прототип:
int fwrite (char *buffer, int size, int count, FILE *stream);

Функцията изисква следните аргументи:
char *buffer – адрес на данните за запис;
int size – размер на всеки един елемент в брой байтове;
int count – максималния брой на данните за запис;
FILE *stream – указател към структура FILE.

Функцията записва ***count*** на брой порции данни, като всяка порция е с размер ***size*** броя байта във файл за запис, определен от потока ***stream.*** Данните за запис се сочат от указателя ***buffer.***
Функцията ***fwrite ()*** връща броя на записаните символи, при нормално приключване, или символ EOF, ако е настъпила грешка.

2.4. Четене на данни от файл

Примери за функции, които осъществяват четене от файл при буфериран достъп са:
fgetc() – чете символ от текущата позиция в потока;
fgets() – чете низ от поток;
fscanf() – чете форматирани данни от поток;
fread() – чете блок данни от поток.

Функцията ***fgetc()*** има следния прототип:
int fgetc(FILE *stream);

Функцията чете символ от текущата позиция от потока, определен от параметъра ***FILE *stream***. Потокът е отворен файл за четене. Функцията връща прочетения символ, при нормално приключване, или символ EOF, ако е настъпила грешка.

Функцията ***fread()*** има следния прототип:
int fread (char *buffer, int size, int count, FILE *stream);

Функцията изисква следните аргументи:
char *buffer – адрес на данните за запис;
int size – размер на всеки един елемент в брой байтове;

int count – максималния брой на данните за запис;
FILE *stream – указател към структура FILE.

Функцията чете максимално ***count*** на брой порции данни, като всяка порция е с размер ***size*** броя байта във файл за четене, определен от потока ***stream.*** Данните, които се четат от файла се записват в паметта на адрес, сочен от указателя ***buffer.***
Функцията ***fread ()*** връща броя на прочетените порции данни. Тази стойност може да е различна от ***count***, при условие, че е настъпила грешка.

III. Контролни въпроси

1. Каква е разликата между буфериран и небуфериран достъп до файлове?
2. Какво е предназначението на функция ***fopen()***?
3. Какво е предназначението на функция ***fclose()***?
4. Чрез кои функции се осъществява запис на данни във файл?
5. Чрез кои функции се осъществява четене на данни от файл?
6. Каква е разликата в начина на достъп при текстови и при двоични файлове?

IV. Задачи за изпълнение

1. Да се състави алгоритъм и програма за мащабиране на затворен контур, описан чрез поредица от точки, като програмата включва следните функции:

- въвеждане от клавиатурата на затворен контур, описан чрез набор точки;
- извеждане на екрана на точките, описващи контура;
- запис на данните във файл;
- четене на данните от файл;
- мащабиране на контур.

Управлението на програмата да се извършва чрез потребителско меню за избор.

Лабораторно упражнение № 15
Динамично разпределение на паметта

I. Цел на лабораторното упражнение

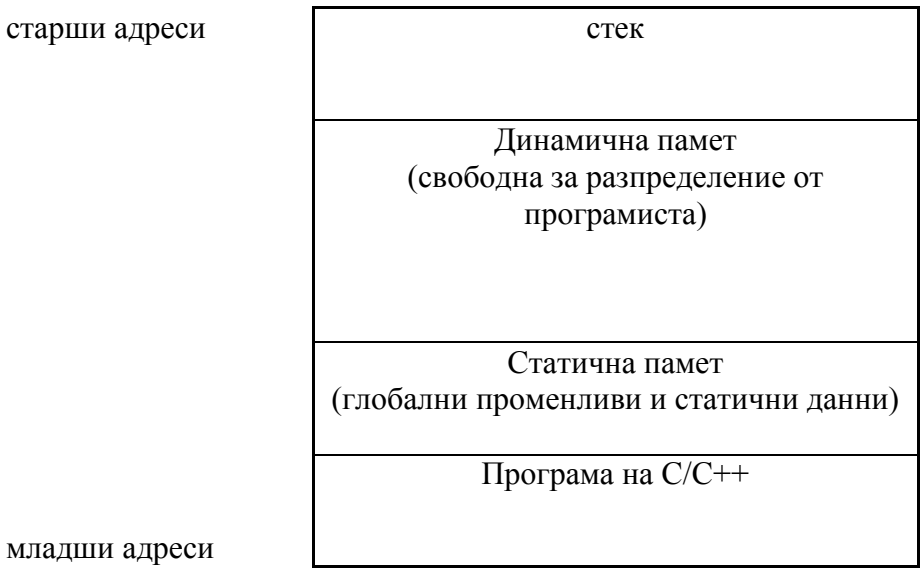
Целта на лабораторното упражнение е запознаване с функциите за динамично разпределение на паметта в ANSI C; създаване на умения за използване в тези функции в различните потребителски програми.

II. Теоретична обосновка

1. Динамична памет

Съвременните езици за програмиране, включително и C++, притежават средства за създаване и унищожаване на компоненти на програма, по време на нейното изпълнение. В случая под компоненти се разбира: променливи, масиви, обекти на класове. Такива компоненти на програмите се наричат ***динамични***.

При стартиране на дадена програма, операционната система отделя необходимия обем памет. За да се използва ефективно, паметта трябва да се разпределя ***динамично***. Това означава, че паметта се разпределя по време на изпълнение на програмата, като се създават и унищожават променливи, масиви, обекти.



Фиг. 15.1. Схема на използване на паметта при изпълнение на програма на C/C++

Според схемата на разпределение на паметта (фиг. 1.1), на младшите адреси се разполага програмния код, следван от статичната памет, която се използва за глобални променливи и статични данни. В старшите адреси се разполага област от паметта, наречена ***стек***. Свободната за динамично разпределение памет се разполага между стека и статичната памет.

2. Оператори *new* и *delete*

Създаването и унищожаването на динамични променливи, масиви, обекти в C++ се извършва от операторите ***new*** и ***delete***. Двата оператора са подобни на библиотечните функции ***malloc()*** и ***free()***.

Общият вид на оператор ***new*** е следния:

new тип [инициализатор]

Съответно:

- **тип** – име на тип, който може да е някой от основните типове или тип, клас, структура, дефинирани от програмиста;
- **инициализатор** – брой елементи, за който се отделя динамична памет; инициализаторът не е задължителен като параметър и ако липсва, се заделя памет само за един елемент.

Оператор ***new*** връща указател към определения тип. Когато не е възможно да се отдели необходимия обем памет, оператор ***new*** връща стойност NULL.

Примери:

```
int *ptr=new int;    // заделя памет за един елемент от тип int
...
char *pch=new char;        // заделя памет за един елемент от тип char
...
const N=10;
int* buffer;
buffer=new int[N];//заделя памет за динамичен буфер от 10 цели числа
if (buffer ==NULL) printf("недостатъчно памет!");
```

Динамично заделената памет, чрез оператор ***new*** не се инициализира и съдържа произволни стойности.

Освобождаването на динамично заделена памет се извършва чрез оператор ***delete***. Общият вид на оператора е следния:

delete указател;

Указателят трябва да сочи динамичен обект, създаден от оператор new.

Примери:

```
delete ptr;   // освобождава памет, заета от указателя ptr
...
delete pch;  // освобождава памет, заета от указателя pch
```

Когато се използва за унищожаване на памет, заделена за масив, оператор ***delete*** има следния общ вид:

delete [] указател;

Пример:

```
int size;
int *buffer;

cin>>size;
if( (buffer=new int[size]) !=NULL)
 { // заделен е динамичен масив от цели числа с размер size
   //...
   delete [] buffer;  // освобождаване на паметта, заета от буфера
 }
else cout<<"недостатъчно памет!";
```

Независимо от това дали оператор ***delete*** съдържа квадратни скоби [] или не, динамично заделения масив ще бъде унищожен т.е. операторът ще освободи заетата памет от всички елементи на масива. Ефектът от наличието на квадратни скоби се проявява тогава, когато елементите на масива са обекти на клас, в който е дефинирана специална член-функция, наречена ***деструктор***. Този метод се изпълнява автоматично при унищожаване на обект. При премахване на масив от обекти, би трябвало да се изпълнят деструкторите на всички елементи на масива. В този случай, ако скобите липсват, ще бъде изпълнен само деструкторът на първия елемент, но не и на останалите, което е грешка при изпълнението на програмата.

3. Функции за динамично разпределение от стандарта UNIX

В език C++ е допустимо използването на функциите от стандарта UNIX за динамично разпределение на паметта. Тези функции могат да се използват вместо оператори ***new*** и ***delete***.

3.1. Динамично отпускане на блок памет

Заделяне на блок памет може да се извърши с функциите ***malloc()*** и ***calloc()***.

Функцията ***malloc()*** има следния прототип:

char *malloc(unsigned size);

Функцията връща указател към заделения блок памет. Параметърът ***size*** определя размера на блока в брой байтове. Ако не може да бъде отделен блок памет с посочения размер, функцията връща стойност NULL.

Функцията ***calloc()*** има следния прототип:

char *calloc(unsigned n, unsigned size);

Функцията заделя блок памет за ***n*** на брой елемента, като всеки елемент е с размер ***size*** байта. Стойностите на елементите в заделения блок се нулират. Размерът на блока в брой байтове е: ***n*size***. Ако не може да бъде отделен блок памет с посочения размер, функцията връща стойност NULL.

3.2. Промяна размера на блок памет

Промяната на размера за блок памет се осъществява чрез функция ***realloc().*** Функцията има следния прототип:

char *realloc(char *ptr, unsigned size);

Функцията променя размера на динамичен блок памет, определен от адреса ***ptr***. Новия размер на блока, в брой байтове, се определя от втория параметър ***size.*** Ако размерът на блока не се намалява, неговото съдържание не се променя. Функцията връща указател към новия блок или стойност NULL, ако промяната на размера на блока не е възможна.

3.3. Освобождаване на блок памет

Преди приключване на програмата, заделената динамична памет трябва да бъде освободена. Освобождаването на блок памет е чрез функция ***free()***. Функцията има следния прототип:

void free(void *ptr);

Функцията освобождава блок памет от адрес ***ptr***, заделен чрез функции ***malloc()***, ***calloc()***, ***realloc().***

3.4. Примери за използване на функциите за динамично разпределение на паметта

В посочената примерна програма се заделя динамичен буфер от **n** цели елемента, като **n** се въвежда от потребителя.

```
#include <stdio.h>
#include <malloc.h>

int main()
{
      int *buffer;
      int n;

      printf("\nBuffer size:");
      scanf("%d",&n);

      buffer=(int*)malloc(n*sizeof(int));
// заделя динамичен масив от n цели елемента

      if (buffer==NULL) printf("Not enough memory!");
// грешка при заделяне на памет
       else
       {       printf("\nBuffer size is %d bytes", n*sizeof(int));
                //...
                free(buffer);  // освобождава заетата памет
       }
      return 0;
}
```

Следващият пример е аналогичен. Отново се заделя динамичен блок памет, като размерът на блока се въвежда от клавиатурата. Чрез използване на функцията ***calloc()***, стойностите в буфера се нулират.

```
#include <stdio.h>
#include <malloc.h>

int main()
{
      int *buffer;
      int n;

      printf("\nBuffer size:");
      scanf("%d",&n);

      buffer=(int*)calloc(n,sizeof(int));
      // заделя динамичен масив от n цели елемента                    // нулира
блока памет
      if (buffer==NULL) printf("Not enough memory!");
      // грешка при заделяне на памет
```

```
    else
    {
            printf("\nBuffer size is %d bytes", n*sizeof(int));
            //...
            free(buffer);           // освобождава заетата памет
    }
   return 0;
}
```

III. Контролни въпроси

1. Кои са предимствата за използването на динамични буфери пред организирането на данните като статични масиви?

2. Каква е схемата на разпределение на паметта при изграждането на програма на C?

3. Какво е предназначението на функция ***malloc()***?

4. Какво е предназначението на функция ***calloc()***?

5. Какво е предназначението на функция ***realloc()***?

6. Какво е предназначението на функция ***free()***?

IV. Задачи за изпълнение

1. Да се състави алгоритъм и програма за транслация на затворен контур, описан чрез поредица от точки, като програмата включва следните функции:

- въвеждане от клавиатурата на затворен контур, описан чрез набор точки;
- извеждане на екрана на точките, описващи контура;
- запис на данните във файл;
- четене на данните от файл;
- транслация на контур.

Управлението на програмата да се извършва чрез потребителско меню за избор. Данните да се съхраняват в паметта в динамичен буфер.

ЛИТЕРАТУРА:

1. Захариева-Стоянова Е., Програмиране и използване на компютри – Програмиране на C/C++, Университетско издателство "Васил Априлов", Габрово, 2017, пето преработено и допълнено издание.
2. Богданов Д., И. Мустакеров, Език за програмиране С, София, Техника, 2000.
3. Езикът C++, София, СофтПрес, 2001.
4. Колисниченко Д., C/C++ практическо програмиране в примери, София, Асеневци, 2017.
5. Кнут Д., Исскуство програмирования для ЭВМ. т.I, Москва, Мир, 1976.
6. Мюлер С., Компютърна енциклопедия, София, СофтПрес, 2002.
7. Наков Св., В. Колев и колектив, Въведение в програмирането на C#, 2015.
8. Симов Г., Програмиране на C++, София, издателска къща СИМ, 1993.
9. Сфар Ч., Microsoft Visual C++ 6.0, София, СофтПрес, 2000.
10. Шилдс Х., C - Практически самоучител, София, СофтПрес, 2001.
11. Bruha I., K.Richta, Programming Language C, Vydavatelstvi CVUT, 1996.

СЪДЪРЖАНИЕ