Символни низове

Структура от данни низ

Логическо описание

Крайна, евентуално празна редица от символи, заградени в кавички, се нарича ***символен низ***, ***знаков низ*** или само ***низ***.

Броят на символите в редицата се нарича **дължина** на низа. Низ с дължина 0 се нарича **празен**.

Примери: "xyz" е символен низ с дължина 3,

"This is a string." е символен низ с дължина 17, а

"" е празния низ.

Низ, който се съдържа в даден низ се нарича негов ***подниз***.

Пример: Низът " is s " е подниз на низа "This is a string.", а низът " is a sing" не е негов подниз.

Конкатенация на два низа е низ, получен като в края на първия низ се запише вторият. Нарича се още **слепване** на низове.

Пример: Конкатенацията на низовете "a+b" и "=b+a" е низът "a+b=b+a", а конкатенацията на "=b+a" с "a+b" е низът "=b+aa+b". Забелязваме, че редът на аргументите е от значение.

Два символни низа се сравняват по следния начин: Сравнява се всеки символ от първия низ със символа от съответната позиция на втория низ. Сравнението продължава до намиране на два различни символа или до края на поне един от символните низове. Ако кодът на символ от първия низ е по-малък от кода на съответния символ от втория низ, или първият низ е изчерпен, приема се, че първият низ е по-малък от втория. Ако пък е по-голям или вторият низ е изчерпен – приема се, че първият низ е по-голям от втория. Ако в процеса на сравнение и двата низа едновременно са изчерпени, те са равни. Това сравнение се нарича ***лексикографско***.

Примери: "abbc" е равен на "abbc"

"abbc" е по-малък от "abbcaaa"

"abbc" е по-голям от "aa"

"abbcc" е по-голям от "abbc".

Физическо представяне

Низовете се представят последователно.

Символни низове в езика C++

Съществуват два начина за разглеждане на низовете в езика C++:

- като едномерни масиви от символи;

- като тип string.

Най-напред ще разгледаме символните низове като масиви от символи.

Дефиницията

```
char str1[100];
```

определя променливата str1 като масив от 100 символа, а

```
char str2[5] = {'a', 'b', 'c'};
```

дефинира масива от символи str2 и го инициализира. Тъй като при инициализацията са указани по-малко от 5 символа, останалите се допълват с нулевия символ, който се означава със символа \0, а понякога и само с 0. Така последната дефиниция е еквивалентна на дефинициите:

```
char str2[4] = {'a', 'b', 'c', '\0'};
```

```
char str2[4] = {'a', 'b', 'c', 0};
```

Този начин за инициализация не е много удобен. Следната дефиниция са еквивалентна на горните.

```
char str[4] = "abc";
```

Забелязваме, че ако низ, съдържащ n символа, трябва да се свърже с масив от символи, минималната дължина на масива трябва да бъде n+1, за да се поберат n-те символа и символът \0.

Друг начин предоставят индексираните променливи.

Примери:

```
q[0] ='a'; q[1] = 's'; q[2] = 'd';
```

При дефиниция на низ с инициализация е възможно size да се пропусне. Тогава инициализацията трябва да съдържа символа '\0' и за стойност на size се подразбира броят на константните изрази, изброени при инициализацията, включително '\0'. Ако size е указано, изброените константни изрази в инициализацията може да са по-малко от size. Тогава останалите се инициализират с '\0'.

Примери:

Дефиницията

```
char q[5] =  {'a', 'b'};
```

е еквивалентна на

```
char q[5]= {'a', 'b', '\0', '\0', '\0'};
```

а също и на

char q[5] = ''ab'';

a

char r[] = {'a', 'b', '\0'}; или

char r[] = ''ab'';

са еквивалентни на

char r[3] = {'a', 'b', '\0'}; или

char r[3] = ''ab'';

Всички действия за работа с едномерни масиви, които описахме в предните теми, са валидни и за масиви от символи с изключение на извеждането. Операторът

cout << str2;

няма да изведе адреса на str2 (както е при масивите от друг тип), а текста

abc

Има обаче една особеност. Ако инициализацията на променливата str2 е пълна (съдържа точно 5 символа) и не завършва със символа \0, т.е. има вида

char str2[5] = {'a', 'b', 'c', 'd', 'e'};

операторът

cout << str2;

извежда текста

abcde<неопределено>

Тип символен низ

Дефиниране

Типът char[size], където size е константен израз от интегрален или изброен тип, ***може да бъде използван за задаване на тип низ с максимална дължина size-1***.

Пример:

char[5] може да се използва за задаване на тип низ с максимална дължина 4.

Множество от стойности

Множеството от стойности на типа низ, зададен чрез char[size], се състои от всички низове с дължина 0, 1, 2, ..., size-1. То е подмножество на множеството от стойности на типа char[size].

Примери:

1. Множеството от стойности на типа низ, зададен чрез char[5] се състои от всички низове с дължина 0, 1, 2, 3 и 4.

Забележка: Редицата {'\0', '\0', '\0', '\0', '\0'} представя празния низ "",{c, '\0', '\0', '\0', '\0'}, където с е произволен символ представя низ с дължина 1, {c, c, '\0', '\0', '\0'} е низ с дължина 2 и т.н.

Елементите от множеството от стойности на даден тип низ са неговите **константи**. Например, "a+b=c-a*e", "1+3", "" са константи от тип char[10].

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип низ, се нарича променлива от този тип низ. Понякога ще я наричаме само низ.

Дефиниция на променливи от тип символен низ

<дефиниция_на_променлива_от_тип_низ> ::=

char <променлива>[size] [= “<редица_от_символи>”|

= {<редица_от_константни_изрази>}]опц

{,<променлива>[size] [= “<редица_от_символи>”|

={<редица_от_константни_изрази>}]опц}опц;

където

- <променлива> е идентификатор;

- size е константен израз от интегрален или изброен тип със положителна стойност;

- редица от константни изрази се дефинира по следния начин:

<редица_от_константни_изрази> ::= <константен_израз>|<константен_израз>, <редица_от_константни_изрази>

като константните изрази в случая са от тип char;

- редица от символи се определя по следния начин:

<редица_от_символи> ::= <празно>| <символ> |<символ><редица_от_символи>

като максималната ѝ дължина е size-1.

Ще отбележим, че фрагментите:

<променлива>[size]

<променлива>[size] = "<редица_от_символи>" и

<променлива>[size] = {<редица_от_константни_изрази>}

могат да се повтарят. За разделител се използва знакът запетая.

Примери:

char s1[5], t[12] = "12345+34";

char s2[10] = "x+y", s4, s5 = {'3', '5', '\0'};

char s3[8] = {'1', '2', '3', '\0'};

Инициализацията е един начин за свързване на променлива от тип низ с конкретна константа от множеството от стойности на този тип низ.

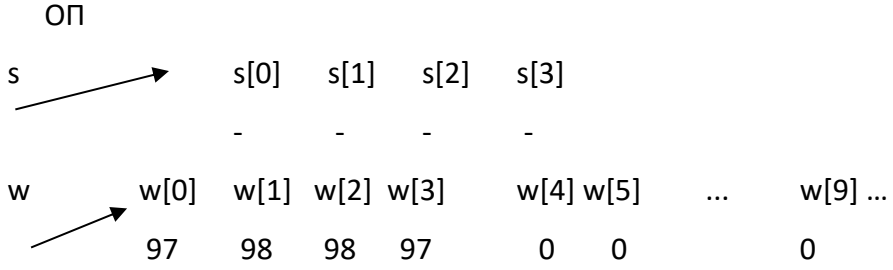
Дефиницията на променлива от тип низ не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет (обикновено 4B), в която записва адреса на първата индексирана променлива, свързана с променливата от тип низ. Останалите индексирани променливи се разполагат последователно след първата. За всяка индексирана променлива се отделя по 1B ОП. Стойността на отделената за индексираните променливи памет е неопределена освен ако не е зададена дефиниция с инициализация. Тогава в клетките се записват инициализиращите стойности, допълнени със знака за край на низ.

Пример: След дефиницията

char s[4];

char w[10] = "abba";

разпределението на паметта има вида:



Операции и вградени функции

Въвеждане на стойност

Реализира се по стандартния начин - чрез оператора cin.

Пример:

char s[5], t[3];

cin >> s >> t;

Настъпва пауза в очакване да се въведат два низа с дължина **не по-голяма** от 4 в първия и не по-голяма от 2 във втория случай. Водещите интервали, табулации и знакът за преминаване на нов ред се пренебрегват. За разделител на низовете се използват интервалът, табулациите и знакът за преминаване на нов ред. Знакът за край на низ автоматично се добавя в края на всяка от въведените знакови комбинации.

Забележка: При въвеждане на низовете не се извършва проверка за достигане на указаната горна граница. Това може да доведе до трудноткриваеми грешки.

Друг начин за въвеждане на низове дава функцията getline.

Извеждане на низ

Реализира се също по стандартния начин – чрез оператора cout.

Пример:

Операторът

```
cout << s;
```

извежда низа, свързан със s. Не е нужно да се грижим за дължината му. Знакът за край на низ идентифицира края му.

Задача 1. Да се напише програма, която въвежда низ, с не повече от 50 символа, завършващ с '@' и проверява дали низът е палиндром.

```
#include <iostream>
#include <stdlib.h>
using namespace std;
char a[50],ch;
int main()
{int n=0,i, n2;
 system("chcp 1251");
 cout<<"въведи низ:\n"; cin>>ch;
 while (ch!='@')
 {a[n]=ch;
  n++;
  cin>>ch;
  }
 n2=n/2;
 for (i=0;i<n2;i++)
  if (a[i]!=a[n-i-1]) break;
 if (i==n2) cout<<a<<" е палиндром"<<endl;
 else cout<<a <<" не е палиндром"<<endl;
 return 0;
 }
```

Преобразуване на низ в цяло число

Реализира се чрез функцията atoi.

Синтаксис

```
atoi(<str>)
```

където <str> е произволен низ (константа, променлива или по-общо израз от тип низ).

Семантика

Преобразува символния низ <str> в число от тип int. Водещите интервали, табулации и знака за преминаване на нов ред се пренебрегват. Символният низ се сканира до първия символ различен от цифра. Ако низът започва със символ различен от цифра и знак, функцията връща 0.

За използване на тази функция е необходимо да се включи заглавният файл stdlib.h.

Примери:

Програмният фрагмент

```
char s[15] = "-123a45";
cout << atoi(s) << "\n";
```

извежда –123, а

```
char s[15] = "b123a45";
cout << atoi(s) << "\n";
```

извежда 0.

Преобразуване на низ в реално число

Реализира се чрез функцията atof.

Синтаксис

atof(<str>)

където <str> е произволен низ (константа, променлива или по-общо израз от тип низ).

Семантика

Преобразува символния низ в число от тип double. Водещите интервали, табулации и знакът за преминаване на нов ред се пренебрегват. Символният низ се сканира до първия символ различен от цифра. Ако низът започва със символ различен от цифра, знак или точка, функцията връща 0.

За използване на тази функция е необходимо да се включи заглавният файл stdlib.h.

Примери:

Програмният фрагмент

```
char st[15] = "-123.35a45";
    cout << atof(st) << "\n";
```

извежда –123.35, а

```
char st[15] = ".123.34c35a45";
cout << atof(st) << "\n";
```

извежда 0.123.

Тип string

В езика за програмиране C++ съществува и друг тип за работа със знакови низове - типа string. Необходимостта от бързо писане на код налага все по-честото използване на този тип.

С него могат да се представят стрингове. Но като цяло има няколко особени предимства пред нормалния `char*`. Първо, размерът му може да се намери чрез метода `.size()`. Второ, не се нуждае от терминираща нула. Трето, позволява лесно разширяване или смаляване (както при вектор). Четвърто, копирането му е много по-

лесно. Пето, предефинирани са няколко допълнителни оператора - например `operator <`, `operator +`, `operator +=`. С тях може да се сравняват и съединяват стрингове сравнително по-лесно, отколкото при char*.

Недостатъците са, че отново, както и при векторите, е малко по-бавен от нормален масив от char.

Деклариране на променлива от тип string

Преди да използваме променливи от тип string е задължително включването на библиотеката <string>, в която е дефиниран типа и всички средства за работа с него.

а) синтаксис:

string <име на променлива>

Тук име на променлива е произволен идентификатор. За разлика от низовете при декларирането на променливи от тип string не е необходимо да определим брой знаци.

б) семантика:

Важно е да отбележим, че типа string сам се грижи за броят на знаците, които ще бъдат записани в него. При първоначалното деклариране се заделя памет само за един знак и по подразбиране това е празният знак. В процеса на въвеждане на повече знаци компилатора автоматично заделя памет за всеки един от тях. Подобно заделяне на памет се нарича динамично. Достъпът до всеки един от елементите на променлива от тип string се осъществява по същият начин както и при масивите от тип char с посочване на името на променливата и индекса на елемента, поставен в скоби.

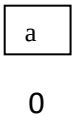
Забележка: Трябва да запомним, че тъй като типа string не е обикновен масив, при първоначалното му деклариране нямаме достъп до произволна клетка от него. Достъп ще имаме само до тези клетки, в които сме записали вече знаци. Записването на знаци в клетките става последователно от ляво на дясно, започвайки от клетка с индекс 0.

Пример:

Нека е декларирана променлива от тип стринг: string s;

Тогава в променливата s на позиция 0 ще бъде записан само празният низ и имаме достъп само до този елемент в момента. Ако приложим операцията присвояване:

s[0]='a' ,



то в нулева позиция ще се запише символа 'a'. Разбира се тази операция може да се приложи за повече от един знак. Например, ако използваме операцията за присвояване по следния начин: s="programming", то в паметта на компютъра ще се задели памет за всичките единадесет знака от низа и ще бъдат записани един след друг в него, както е показано на фигурата:

s

p	r	o	g	r	a	m	m	i	n	g
0	1	2	3	4	5	6	7	8	9	10

В този случай имаме достъп само до елементите от индекс 0 до индекс 10.

Въвеждане на променлива от тип string

а) използване на операторите за вход и изход от библиотеката iostream

За операторите cin и cout е предвидена възможност за въвеждане и извеждане на променливи от тип string, така че за разлика от числовите масиви те могат да се въвеждат от клавиатурата само с един оператор като прости променливи.

Например за въвеждането от клавиатурата на декларираната по-горе променлива

можем да използваме следния оператор за `cin»s;`

и да я изведем на екрана

`cout«s;`

б) въвеждане на ЦЯЛ ред от текст

Да си припомним, че при въвеждане на знаци със cin се пропускат интервалите и знаците за край на ред и край на текст. При въвеждане на променливи от тип string тези знаци означават край на четения текст и в него се записват само знаците, въведени преди тях. Понякога може да се наложи в низа да бъдат записани няколко, а не само една дума, например, цял ред. Тогава можем да използваме оператора-функция getline, който има следния синтаксис:

getline(cin, <идентификатор>, <знак>)

Първият параметър показва че текстът се чете от клавиатурата. Като втори параметър се посочва името на променливата. Последният параметър показва кой знак ще се използва за прекъсване на четенето на низа. Обикновено, ако искаме да прочетем цял ред, се използва знака '\n' . Възможно е третия параметър да бъде пропуснат като по подразбиране се приема, че за край на четенето ще бъде сложен знака '\ n' .

Примери:

```
string s;
 getline(cin, s, '.');
 cout<<s<<endl;
```

Ако въведем: **Това е само. пример**
Резултатът е: **Това е само**

```
string s;
 getline(cin, s, '\n');
 cout<<s<<endl;
```

Ако въведем: **Това е само пример. Още веднъж**
Резултатът е: **Това е само пример. Още веднъж**

Сравняване на променливи от тип string.

За разлика от низовете, където има функция за сравняването им, при променливите от тип string можем да използваме познатите ни вече оператори за сравнение <, >, <=, >=, ==. Ще припомним отново, че сравняването на два низа става лексикографски.

Ето и следните няколко примера:

string s1= "aab", s2 = "aьa";

то резултатите от следните няколко сравнения са:

(s1>s2) 0-лъжа
(s1<s2) 1-истина
(s1>= s2) 0-лъжа
(s1<=s2) 1-истина
(s1==s2) 0-лъжа
(s1==s1) 1-истина
Резултатите от операторите за сравнение са само 1 (истина) и 0 (лъжа).

Задача 3. Напишете програма, която въвежда текст, завършващ с '*' и заменя всяко срещане на дума с друга дума.

```
#include <iostream>
#include <string.h>
#include <stdlib.h>
using namespace std;
char c;
string ch,a,b;
int main()
{
 system("chcp 1251");
 cout<<"търси:";cin>>a;
 cout<<"замени с:";cin>>b;
 cin>>ch;cin.get(c);//четем дума след това и интервала
 while(ch[ch.length()-1]!= '*') //докато свърши текста
  {
    if (a==ch) cout<<b<<c;
    else cout<<ch<<c;
    cin>>ch;cin.get(c);//четем дума след това и интервала
   }
  cout<<endl;
 }
```

Присвояване на стойност на променлива от тип string

Както по-горе вече показахме присвояване на стойност на променлива от тип string може да стане с оператора за присвояване =. Можем да присвоим както произволен текст с една, две и т.н. символа, така и стойност на друга променлива от тип string.

Пример:

s1=s2;

s1="aЬbcЬa";

Забележка: При присвояване на стойност на дадена променлива нейното съдържание се изтрива и на негово място се записва присвоената стойност без значение на нейната дължина. Низът приема новата стойност и паметта, заделена за него, се променя в зависимост от дължината ѝ.

Слепване на низове (конкатенация)

Можем да залепим съдържанието на една променлива от тип string за съдържанието на друга променлива от тип string с помощта на оператора +. При това ние сами можем да определим къде да бъде записано залепеното съдържание. То може да бъде залепено и записано в допълнителна променлива, но е възможно и да бъде залепено в някоя от първоначално използваните две. За второто действие е необходим оператора +=. Резултатът от изпълнението на операциите + и += става по-ясен от следващия пример.

Пример

```
s1="abc";
s2="cde";
s=s1+s2; //s=="abccde"
s1+=s2;//s1=="abccde"
s2+= s1 //s2=="cdeabc"
```

Задача 4. Да се напише програма, която въвежда текст, завършващ с `'*'` премахва символа `'s'` и извежда новия текст.

```
#include <iostream>
#include <string>
using namespace std;
int  main()
{
 char  c;string a="";
 do
 {cin>>c;
  if (c!='s')
  a+=c;
  }
 while (c!='*');
```

```
 cout<<a;
}
```

Функции за работа с променливи от тип string, поддържани от библиотеката string.

а) определяне на брой знаци в низа (размер на низа)

s .length () s.size()

Двете функции връщат като резултат броя на знаците в низа s.

Задача 5. Напишете програма, която въвежда текст, завършващ с '*' и отпечатва думите започващи и завършващи с 'a'.

```
#include <string.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
int main()
{
 string ch;
 int n;
 system("chcp 1251");
 cin>>ch; n=ch.length()-1;
 while(ch[n]!= '*')
  {
    if (ch[0]=='a' && ch[n]=='a') cout<<ch<<endl;
    cin>>ch;n=ch.length()-1;
   }
  }
```

б) отделяне на част от низа

s. substr (n1, n2) - отделя n2 на брой знака от низа, започвайки от позицията с номер n1.

```
string s="This is a text";
cout<<s.substr(3,6)<<endl;
```

s is a

в) изтриване на съдържанието на целия низ

s. clear () - след изпълнението на тази функция низът не съдържа знаци и дължината му е 0.

г) изтриване на част от низ

s. erase (n1, n2) - изтрива n2 на брой знака от низа, започвайки от позицията с номер n1.

```
string s="This is a text";
cout<<s.erase(4,6)<<endl;
```

Thistext

д) вмъкване на друг низ в дадения.

s. insert (n, s1) - вмъква низа s1 в низа s от позиция с номер n.

```
string s="This is a text";
cout<<s.insert(10,"simple ")<<endl;
```

This is a simple text

е) размяна на два низа.

s . swap (s1) - разменя стойностите на низа s и s1.

```
string s="This is a text", s1="simple ";
s.swap(s1);
cout<<s<<endl;
cout<<s1<<endl;
```

simple

This is a text

ж) проверка за празен низ.

s . empty () - връща стойност "истина", ако низа s е празен и "лъжа" в противен случай.

з) проверка дали определен низ се съдържа в дадения.

s. find (str, pos) - проверява дали низа str, зададен като параметър се съдържа в s.

Параметърът pos, който по подразбиране е 0 означава, че от str може да се вземе само тази част, която започва от pos и да се проверява дали тя е подниз на s. Ако параметър не се зададе, се подразбира, че ще се търси целия низ str. Резултатът от функцията е позицията на първото срещане на str в s.

```
string s="This is a text";
cout<<s.find("text")<<endl;
```

10

и) `s.begin()` - връща указател към началото на стринга s.

`s.end()` – връща указател към края на стринга s.

Проследявайки изпълнението на задачи 3 и 5, прави впечатление , че въпреки верния резултат, той не е много удобен за четене – редуват с е ред с входни данни и ред с изходни резултати. В такива случай е удобно данните да се въвеждат от текстов файл. В следващата лекция ще разгледаме как може да се използват текстовите файлове.

Текстови файлове

Основни понятия. Стандартни потоци за вход и изход

При въвеждане и извеждане на данни в езика C++ често се споменава понятието „поток” (stream). Потокът е последователност от байтове, които „текат” в една посока (при четене – вход; при запис - изход).

Файловите операции (четене и запис) могат да се реализират, чрез потоци, които са обекти на класовете fstream, ifstream и ostream, които са съответно за четене/запис, четене и запис.

Във входно-изходните библиотеки класовете са няколко на брой:

- ***ios*** – базов клас, съдържащ потоци;
- ***istream*** – потоци за вход;
- ***ostream*** – потоци за изход;
- ***iostream*** – потоци за вход и изход;
- ***ifstream*** – потоци за вход от файл;
- ***ofstream*** – потоци за изход към файл;
- ***fstream*** – потоци за вход и изход, свързани с файл.

Двата най-често използвани потока ***cin*** и ***cout*** са тези за стандартния вход и изход, т.е. въвеждане от клавиатура и извеждане на екрана.

Потокът ***cin*** е клас от библиотеката ***istream***. Той се свързва с въвеждане на входни данни от клавиатурата:

cin>>i_1>>i_2>>…>>i_n;

Операцията >> се използва за четене на цели и реални числа, както и на символни низове. При четене на повече от една стойности между тях трябва да има поне един празен интервал, табулатор или символ за нов ред.

Потокът ***cout*** е клас от библиотеката ***ostream***. Той се свързва с извеждане на данни на екрана:

cout<<i_1<<i_2<<…<<i_n;

Операцията << се използва за извеждане на цели и реални числа, както и на символни низове. При извеждане на повече от една стойност при нужда програмистът трябва да се погрижи за извеждането на разделители между тях, например: ***'\n'*** (нов ред или ***endl***), ***'\t'*** (табулатор), или един или повече интервали.

Ако в текста, който се извежда, трябва да бъдат използвани кавички, това става чрез \".

Аналогично, ако желаете в символния низ да има знак \, използвайте две наклонени черти \\.

Двата стандартни потока ***cin*** и ***cout*** са дефинирани в заглавния файл (библиотека) ***iostream.h***. Затова програмите, в които има въвеждане на данни от клавиатура и извеждане на данни на екрана, трябва да съдържат директивата към препроцесора:

#include <iostream>

Файлове и оператори за работа с потоци

Определение за файл: *Крайна последователност от байтове, записана на външен носител и завършваща със знак за край на файл. В C++ байтовете са номерирани последователно, започвайки от 0.*

Тук се разглеждат само файлове с т.н. последователен достъп (текстови файлове).

В програмите често се налага входните данни да бъдат прочетени от файл, или получените резултати да бъдат записани във файл. Концепцията за потоците е приложена и при работа с файлове. Езикът C++ поддържа обекти от типа ***ifstream*** (входен файлов поток) и ***ofstream*** (изходен файлов поток). За да могат да бъдат използвани тези два потока, в началото на програмата трябва да се включи заглавния файл ***fstream*** като директива към препроцесора:

#include <fstream>

Този заглавен файл автоматично включва в себе си ***iostream.***

За да се създадат обекти от тип файлов поток (файлови променливи), се използват оператори от типа:

ifstream fin("c:\\data\\in-data.txt");
ofstream fout("c:\\data\\out-data.txt");
fin и ***fout*** са примерни имена на файлови потоци.
Както по-горе беше споменато, компилаторът преобразува двойката наклонени черти \\ в една \, която се използва при задаването на директории.

След дефинирането на ***fin*** и ***fout*** те могат да бъдат използвани по подобие на ***cin*** и ***cout***.

Обектите потоци връщат като резултат нулева стойност ***NULL*** (***'\0'***), ако е възникнала грешка при четене, например е достигнат краят на файла.

Името на файла може да бъде въведено и от клавиатурата:

```
char  filename[40];
cout<<"Въведете име на файл: "; cin>>filename;
ofstream f_out(filename);
if (f_out == NULL)
    cout <<"Грешка при отваряне на файла.\n" ;
else
        cout<<"OK\n";
```

Ред на работа

- отваряне на файла за четене или запис – файловата променлива се свързва с физически файл от външната памет и се определя режима на достъп до него.

- четене от файла или запис в него – чрез файловата променлива се осъществява буфериран достъп до данните от външния файл.

- затваряне на файла – буферите се записват във външния файл и връзката между него и файловата променлива се прекъсва.

Деклариране на файл

ofstream <име> ; // файл само за запис
ifstream <име>; // файл само за четене
fstream <име>; // файл за четене и запис
- <име> – името на файловата променлива
- ofstream – потребителски тип за връзка с файл за запис, дефиниран в fstream.h
- ifstream – тип за връзка с файл за четене
- fstream – тип за файл за четене и запис

Отваряне на файла

Предварително се дефинира обектът – поток, а по-късно се задава името на файла и режимът на достъп до него. Ето един пример за отваряне на файл с цел четене:

ifstream <променлива>***;***

<променлива>.open(<файл>, <режим>);
- <променлива>– името на файловата променлива
- <файл> – името на потребителския файл
- <режим>– указва режима на четене или запис.

Пример:

MyFile.open("test.dat", ios::in);

NewFile.open("new.dat", ios::in | ios::out);

Режими на отваряне на файла
ios::in – отваря файла за четене
ios::out – отваря файла за запис, като изтрива съдържанието му, ако файла съществува
ios::app – отваря файла за добавяне само в края на файла. Съдържанието му се запазва.
ios::ate – отваря файла за запис и премества указателя за запис в края на файла. Указателят може да бъде преместван на произволни места.
ios::trunc – унищожава предишното съдържание на файла;
ios::binary - запис и четене на данни в т.н. бинарен режим.
Файлът, който се отваря за четене, трябва да съществува.

Има и друг начин за отваряне на файл.

При отваряне на файл само за запис използваме конструкцията:

ofstream <име_файлов_поток>(“име_файл”, ios::out);

Ако файлът със зададеното име не съществува, той бива създаден. Ако съществува, се нулира съдържанието му и записът започва отначало.

При отваряне на файл само за четене използваме конструкцията:

ifstream <име_файлов_поток>(“име_файл”, ios::in);

Освен параметрите ***in*** и ***out*** на ***ios***, съществуват и други, които се задават с побитово ИЛИ (отбелязва се с единична вертикална черта:|):

Четене и запис във файл

<променлива> << израз; // запис
<променлива> >> променлива; // четене
<променлива> – файловата променлива
Пример:
MyFile<< "Proba\n\n";
NewFile >> p;

Затваряне на файл

<променлива>.close();
<променлива> – файловата променлива
пример:
MyFile.close();
Последният оператор затваря отворения файл.

Операции с файлове

Основните операции с файлове могат да се разделят на 4 групи:

- операции за създаване на файл;
- операции за отваряне и затваряне на файл;
- операции за избор на режим за работа с файл;
- операции за позициониране във файл.

Методи с файлови потоци:

- io.is_open(); - проверява дали файлът е отворен
- io.close(); - затваря файла
- io.eof(); - проверява дали сме достигнали края на файл-а
- io.get(ch); - чете символа ch
- io.put(ch); - записва символа ch

Задача 3 (от предната тема)**.** *Да се напише програма, която в текстов файл заменя всяко срещане на дума с друга дума.*

```
#include <iostream>
#include <fstream>
```

```
#include <string.h>
#include <stdlib.h>
using namespace std;
ifstream inf;//създаване на входен поток
ofstream outf;//създаване на изходен поток
char c;
string ch,a,b;
int main()
{
 char st[10];
 system("chcp 1251");
 cout<<"търси:";cin>>a;
 cout<<"замени с:";cin>>b;
 cout<<"име на файл:";cin>>st;
 inf.open(st, ios::in);//отваряне на файла
 outf.open("proba2.txt", ios::out);
 inf>>ch;inf.get(c);//четем от файла дума след това и
интервала
 while(inf) //докато свърши файла
  {
    if (a==ch) outf<<b<<c;//запис в новия файл
    else outf<<ch<<c;//запис в новия файл
    inf>>ch;inf.get(c);//четем от файла дума след това и
интервала
   }
 inf.close();outf.close();
 ifstream inf; //за да изведем на екрана файла създаваме
входен поток
 inf.open("proba2.txt", ios::in); //свързваме го с новия
файл
 while(inf) //  извеждаме го на екрана
  { inf.get(c); cout<<c;}
    inf.close();
    cout<<endl;
   }
```

Задача 5 (от предната тема)**.** Да се напише програма, която в текстов файл с дадено име отпечатва думите започващи и завършващи с 'а'.

```
#include <fstream>
……
int main()
{ ifstream inf;
  string ch; char st[10];
  ……..
  cout<<"име на файл:";cin>>st;
  inf.open(st, ios::in);
  inf>>ch;
  while(inf)
   {
    …….
     inf>>ch;
```

```
 }
inf.close();
return 0;
}
```

Рекурсия

1. Рекурсивни функции в математиката

Ако в дефиницията на някаква функция се използва самата функция, дефиницията на функцията е рекурсивна.

Примери:

а) Ако n е произволно естествено число, следната дефиниция на функцията факториел

е рекурсивна

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

Условието при n = 0 не съдържа обръщение към функцията факториел и се нарича гранично.

б) Функцията за намиране на най-голям общ делител на две естествени числа a и b може да се дефинира по следния рекурсивен начин:

$$\gcd(a,b)=\begin{cases}a, & a=b\\ \gcd(a-b,b), & a>b\\ \gcd(a,b-a), & b>a\end{cases}$$

Тук граничното условие е условието при a = b.

в) Редицата от числата на Фибоначи

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

може да се дефинира рекурсивно по следния начин:

$$fib(n)=\begin{cases}1, & n=0\\ 1, & n=1\\ fib(n-1)+fib(n-2), & n>1\end{cases}$$

В този случай имаме две гранични условия – при n = 0 и при n = 1.

Рекурсивната дефиниция на функция може да се използва за намиране стойността на функцията за даден допустим аргумент.

Примери:

а) Като се използва рекурсивната дефиниция на функцията факториел може да се намери факториелът на 4. Процесът за намирането му преминава през разширение, при което операцията умножение се отлага, до достигане на граничното условие 0! = 1. Следва свиване, при което се изпълняват отложените операции.

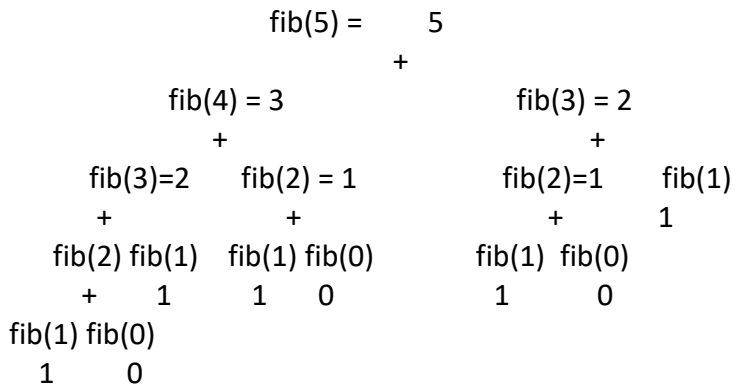
4! =
4.3! =
4.3.2! =
4.3.2.1! =
4.3.2.1.0! =
4.3.2.1.1 =
4.3.2.1 =
4.3.2 =
4.6 =
24

б) Като се използва рекурсивната дефиниция на функцията gcd, може да се намери gcd(35, 14).

gcd(35, 14) =
gcd(21, 14) =
gcd(7, 14) =
gcd(7, 7) =
7.

В този случай няма разширяване, нито пък свиване. Двата аргумента на gcd играят ролята на натрупващи параметри. Те се променят докато се достигне граничното условие.

в) Чрез рекурсивната дефиниция на функцията, генерираща редицата на Фибоначи, може да се намери fib(5).



В този случай намирането на fib(5) генерира дървовиден процес.

Операцията + се отлага. Забелязваме много повтарящи се изчисления, например fib(0) се пресмята 3 пъти, fib(1) – 5 пъти, fib(2) - 3 пъти,fib(3) – 2 пъти, което илюстрира неефективността на този вид пресмятане.

2. Рекурсивни функции в C++

Известно е, че в тялото на всяка функция може да бъде извикана друга функция, която е дефинирана или е декларирана до момента на извикването ѝ. Освен това, в C++ е вграден т. нар. механизъм на рекурсия – разрешено е функция да вика в тялото си самата себе си.

Функция, която се обръща пряко или косвено към себе си, се нарича рекурсивна. Програма, съдържаща рекурсивна функция е рекурсивна.

Чрез пример ще илюстрираме описанието, обръщението и изпълнението на рекурсивна функция.

Задача 1. *Да се напише рекурсивна програма, която въвежда естествено число* ***m*** *и намира* ***m!.***

```
//Program Zad1.cpp
#include <iostream>
using namespace std;
int factorial(int);
int main()
{cout << "m= ";
 int m;
 cin >> m;
 if(!cin || m < 0)
  {cout << "Error! \n";
   return 1;
   }
 cout << m << "!= " << factorial(m) << '\n';
 return 0;
}
int factorial(int n)
{if (n == 0) return 1;
 else return n * factorial(n-1);
 }
```

В тази програма е описана рекурсивната функция `factorial`, която приложена към естествено число, връща факториела на това число. Стойността на функцията се определя посредством обръщение към самата функция в оператора `return n * factorial (n-1);`.

Изпълнение на програмата

Изпълнението започва с изпълнение на главната функция. Фрагментът

```
cout << "m= ";
int m;
cin >> m;
```

въвежда стойност на променливата m. Нека за стойност на m е въведено 4. В резултат в стековата рамка на main, отделените 4В за променливата m се инициализират с 4. След това се изпълнява операторът

```
cout << m << "!= " << factorial (m) << '\n';
```

За целта трябва да се пресметне стойността на функцията `factorial(m)` за m равно на 4, след което получената стойност да се изведе. Обръщението към функцията `factorial` е илюстрирано по-долу:

Генерира се стекова рамка за това обръщение към функцията `factorial`. В нея се отделят 4В ОП за формалния параметър n, в която памет се откопира стойността на фактическия параметър m и започва изпълнението на тялото на функцията. Тъй като n е различно от 0, изпълнява се операторът

```
return n * factorial (n-1);
```

при което трябва да се намери `factorial(n-1)`, т.е. `factorial(3)`. По такъв начин преди завършването на първото обръщение към `factorial` се прави второ обръщение към тази функция. За целта се генерира нова стекова рамка на функцията `factorial`, в която за формалния параметър n се откопира стойност 3. Тялото на функцията `factorial` започва да се изпълнява за втори път (Временно спира изпълнението на тялото на функцията, предизвикано от първото обръщение към нея).

По аналогичен начин възникват още обръщения към функцията `factorial`. При последното от тях, стойността на формалния параметър n е равна на 0 и тялото на `factorial` се изпълнява напълно. Така се получава:

m 4 0x0066FDF4 стекова рамка на main
…….
n 4 0x0066FDA0 стекова рамка на 1-во обръщение към `factorial`
……
n 3 0x0066FD48 стекова рамка на 2-ро обръщение към `factorial`
……
n 2 0x0066FCF0 стекова рамка на 3-то обръщение към `factorial`
……..
n 1 0x0066FC98 стекова рамка на 4-ро обръщение към `factorial`
……..
n 0 0x0066FC40 стекова рамка на 5-то обръщение към `factorial`
……

При петото обръщение към `factorial` стойността на n е равна на 0. В резултат, изпълнението на това обръщение завършва и за стойност на `factorial` се получава 1. След това последователно завършват изпълненията на останалите обръщения към тялото на функцията. При всяко изпълнение на тялото на функцията се определя съответната стойност на функцията `factorial`. След завършването на всяко изпълнение на функцията `factorial`, отделената за `factorial` стекова рамка се освобождава. В крайна сметка в главната функция се връща 24 – стойността на 4!, която се извежда върху екрана. В този случай, рекурсивното дефиниране на функцията факториел не е подходящо, тъй като съществува лесно итеративно решение.

Задача 2. *Да се напише рекурсивна програма, която въвежда естествено число **m** и намира **m**-то на Фибоначи.*

```
#include <iostream>
using namespace std;
int fib(int n)
{if (n == 0 || n == 1 ) return 1;
 else return fib(n-1)+fib(n-2);
 }

int main()
{cout << "m= ";
 int m;
```

```
 cin >> m;

 cout << fib(m) << '\n';
 return 0;
}
```

Както отбелязахме и по-горе, пресмятането на числата на Фибоначи генерира дървовиден процес с многократни пресмятания на едни и същи изрази.

Препоръка: Ако за решаването на някаква задача може да се използва итеративен алгоритъм, реализирайте го. Не се препоръчва винаги използването на рекурсия, тъй като това води до загуба на памет и време.

Съществуват обаче задачи, които се решават трудно, ако не се използва рекурсия.

Задача 3. *Да се напише програма, която въвежда редица от числа, завършваща с 0 и ги извежда в обратен на въвеждането им ред. (Заб. Да не се използва масив)*

```
#include <iostream>
using namespace std;
void  obratno()
{ int x;
  cin>>x;
  if (x!=0) { obratno();cout<<x<<"  ";}
 }
int main()|
 { obratno();cout<<endl;
  }
```

3. Бързо сортиране

Може би си спомняте, че в първата лекция разгледахме няколко метода за сортиране на масиви и споменахме, че ще разгледаме един подобрен метод на метода на мехурчето. Този метод се оказва най-бързия метод за сортиране, откъдето идва и името му - quick sort. Сложността на този метод е O(n.log(n)).

Идеята на този метод е да се избере елемент от масива и спрямо него, масивът да се раздели на два дяла – отляво по-малките от избрания, а отдясно по-големите. Върху тези два дяла се прилага същото действие и така, докато дяловете станат от по 1 елемент. Самото описание на алгоритъма предполага реализация с рекурсия.

19 5 14 9 17 3 1 15
1 5 14 9 17 3 19 15
1 5 3 9 17 14 19 15

1 5 3 9
1 3 5 9

17 14 19 15
17 14 15 19

```
#include <iostream>
using namespace std;
long a[10000000]; int n;
void sort(int l,int r)
 {
  int i,j;long x,y;
   i=l; j=r; x=a[(l+r)/ 2];
  do
   { while (x>a[i]) i++;
     while (a[j]>x) j--;
     if (i<=j){
              y=a[i]; a[i]=a[j]; a[j]=y;
              i++; j--;
          }
   }
   while (i<=j);
   if (l<j) sort(l,j);
   if (i<r) sort(i,r);
  }
int main()
{
 cin>>n;
 for (int i=0;i<n;i++)
 cin>>a[i];
 sort(0,n-1);
 for (int i=0;i<n;i++)
 cout<<a[i]<<" ";
 return 0;
}
```

***4*. Взаимна рекурсия**

Може една функция да се обърне към друга, а втората към първата. Това се нарича *взаимна рекурсия.*

Задача 4. *Да се състави програма за пресмятане на функцията A по формулите:*

$A(n)=\begin{cases}1, & n=1\\ 2*B(n-1), & n\neq 1\end{cases}$

$B(n)=\begin{cases}1, & n=1\\ 2*A(n-1), & n\neq 1\end{cases}$

```
#include<iostream>
using namespace std;
int a(int);
int b(int n)
{if(n==1)return 1;
 else return 2+a(n-1);
}
int a(int n)
{if(n==1)return 1;
 else return 2*b(n-1);
```

```
}
int main()
{
 cout<<a(5)+b(3)<<endl;
}
```

Забелязваме простотата и компактността на записа на рекурсивните функции. Това проличава особено при работа с динамичните структури: свързан списък, стек, опашка, дърво и граф.

Основен недостатък е намаляването на бързодействието поради загуба на време за копиране на параметрите им в стека. Освен това се изразходва повече памет, особено при дълбока степен на вложеност на рекурсията.

Задача 5. *Ханойски кули.* Има три пилона. На един от тях са наредени един върху друг дискове с различен диаметър. Искаме на преместим дисковете на някой от другите пилони, спазвайки следните правила: може да се премества само по един диск; не може да се постави диск с по-голям диаметър върху такъв с по-малък диаметър. Да се напише програма, която въвежда брой дискове и имена на пилони и отпечатва преместванията на дисковете, спазвайки горните правила.

```
#include <iostream>
#include <stdlib.h>
using namespace std;
void move(int n, char begin, char end)
{
 cout<<"премести "<<n<<"-тия диск от "<<begin
<<" на "<<end<<endl;
}
void hanoy(int n, char begin, char temp, char end)
{if(n==1)move(n,begin,end);
 else {hanoy(n-1,begin, end,temp);
       move(n,begin,end);
       hanoy(n-1,temp, begin, end);
         }
}
int main()
{char b,t,e;int n;
 system("chcp 1251");
 cout<<"брой дискове:"; cin>>n;
 cout<<"въведи имена на пилоните(начален, помощен, краен):";
 cin>>b>>t>>e;
 hanoy(n,b,t,e);
}
```

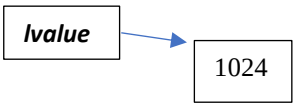
Типове указател и псевдоним

Тип указател

Променлива, това е място за съхранение на данни, което може да съдържа различни стойности. Идентифицира се с дадено от потребителя име (идентификатор). Има си и тип. Дефинира се като се указват задължително типа и името ѝ. Типът определя броя на байтовете, в които ще се съхранява променливата, а също и множеството от операциите, които могат да се изпълняват над нея. Освен това, с променливата е свързана и стойност–неопределена или константа от типа, от който е тя. Нарича се още ***rvalue***. Мястото в паметта, в което е записана ***rvalue*** се нарича адрес на променливата или ***lvalue***. По-точно адресът е адреса на първия байт от множеството байтове, отделени за променливата.

Пример: Фрагментът

int i = 1024;



дефинира променлива с име i и тип int. Стойността ѝ (rvalue) е 1024. i именува място от паметта (lvalue) с размери 4 байта, като lvalue е адреса на първия байт.

Намирането на адреса на дефинирана променлива става чрез унарния префиксен дясноасоциативен оператор & (амперсанд). Приоритетът му е същия като на унарните оператори +, -, !, ++, -- и др.

Синтаксис

&<променлива>

където <променлива> е вече дефинирана променлива.

Семантика

Намира адреса на <променлива>.

Пример:&i е адреса на променливата i и може да се изведе чрез оператора cout <<&i;

Операторът & не може да се прилага върху константи и изрази, т.е. &100 и &(i+5) са недопустими обръщения. Не е възможно също прилагането му и върху променливи от тип масив, тъй като те имат и смисъла на константни указатели. Адресите могат да се присвояват на специален тип променливи, наречени променливи от тип указател или само указатели.

Задаване на тип указател

Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен указател към T. T се нарича **указван тип или тип на указателя**.

Примери:

int* е тип указател към int;

Множество от стойности

Състои се от адресите на данните от тип T, дефинирани в програмата, преди използването на T*. Те са константите на типа T*.

Освен тях съществува специална константа с име NULL, наречена нулев указател. Тя може да бъде свързвана с всеки указател независимо от неговия тип. Тази константа се интерпретира като “сочи към никъде”, а в позиция на предикат е false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа T, се нарича променлива от тип T* или променлива от тип указател към тип T. Дефинира се по стандартния начин.*

Дефиниция на променлива от тип указател

T* <променлива> [= <стойност>]; |T *<променлива> [= <стойност>];

където

T е име или дефиниция на тип;
<променлива> ::= <идентификатор>
<стойност> е шестнадесетично цяло число, представляващо адрес на данна от тип T или NULL.

T определя типа на данните, които указателят адресира, а също и начина на интерпретацията им.

Възможно е фрагментите

<променлива> [=<стойност>] и
*<променлива>[=<стойност>]

да се повтарят. За разделител се използва запетаята. В първия случай обаче има особеност. Дефиницията

T* a, b;

е еквивалентна на

T* a;

T b;

т.е. само променливата a е указател.

Примери: Дефиницията

int *pint1, *pint2;

задава два указателя към тип int, а

int* pint1, pint2;

указател pint1 към int и променлива pint2 от тип int.

Дефиницията на променлива от тип указател предизвиква в ОП да се отделят 4В, в които се записва някакъв адрес от множеството от стойности на съответния тип, ако дефиницията е с инициализация и неопределено или NULL, ако дефиницията не е с инициализация. Този адрес е стойността на променливата от тип указател, а записаното на този адрес е съдържанието ѝ.

Пример:

```
#include <iostream>
using namespace std;
int main()
{int i=12; int* p=&i;
double *q=NULL;
double x=1.56;
double *r=&x;
cout<<i<<"  "<<*p<<" "<<p<<endl;
cout<<x<<"  "<<*r<<"  "<<q<<" "<<r<<endl;
*p=20;*r=2.18;
cout<<i<<"  "<<x<<endl;
}
```

Дефинициите

Int i = 12;

int* p = &i;// p е инициализирано с адреса на i

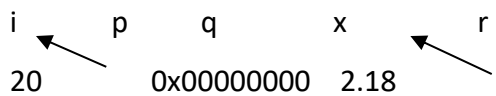
double *q = NULL; // q е инициализирано с нулевия указател

double x = 1.56;

double *r = &x;// r е инициализирано с адреса на x

предизвикват следното разпределение на паметта

ОП



Съвет: Всеки указател, който не сочи към конкретен адрес, е добре да се свърже с константата NULL. Ако по невнимание се опитате да използвате нулев указател, програмата ви може да извърши нарушение при достъп и да блокира, но това е по-добре, отколкото указателят да сочи към кой знае къде.

Операции и вградени функции

Извличане на съдържанието на указател

Осъществява се чрез префиксния, дясноасоциативен унарен оператор * .

Синтаксис

*<променлива_от_тип_указател>

Семантика

Извлича стойността на адреса, записан в <променлива_от_тип_указател>, т.е. съдържанието на <променлива_от_тип_ указател>.

Като използваме дефинициите от примера по-горе, имаме:

*p е 12// 12 е съдържанието на p
*r е 1.56// 1.56 е съдържанието на r

Освен, че намира съдържанието на променлива от тип указател, обръщението

*<променлива_от_тип_указател>

е данна от тип T (променлива или константа). Всички операции, допустими за типа T, са допустими и за нея.

Като използваме дефинициите от примера по-горе, *p и *r са цяла и реална променливи, съответно. След изпълнение на операторите за присвояване

*p = 20;
*r = 2.18;
стойността на i се променя на 20, а тази на x – на 2.18.

Аритметични и логически операции

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, >, >=, < и <=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още адресна аритметика. Особеността се изразява в т. нар. мащабиране. Ще го изясним чрез пример.

Да разгледаме фрагмента

```
int *p;
double *q;
...
p = p + 1;
q = q + 1;
```

Операторът p = p+ 1; свързва p не със стойността на p, увеличена с 1, а с p + 1*4, където 4 е броя на байтовете, необходими за записване на данна от тип int (p е указател към int). Аналогично, q = q + 1; увеличава стойността на q не с 1, а с 8, тъй като q е указател към double (8 байта са необходими за записване на данна от този тип).

Общото правило е следното: Ако p е указател от тип T, p+i е съкратен запис на p + i*sizeof(T), където sizeof(T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T.*

Въвеждане

Не е възможно въвеждане на данни от тип указател чрез оператора cin. Свързването на указател със стойност става чрез инициализация или оператора за присвояване.

Извеждане

Осъществява се по стандартния начин – чрез оператора cout.

Допълнение

Типът, който се задава в дефиницията на променлива от тип указател, е информация за компилатора относно начина, по който да се интерпретира съдържанието на указателя. В контекста на горния пример *p са четири байта, които ще се интерпретират като цяло число от тип int. Аналогично, *q са осем байта, които ще се интерпретират като реално число от тип double.

В някои случаи е важна стойността на променливата от тип указател (адресът), а не нейното съдържание. Тогава тя се дефинира като указател към тип **void**. Този тип указатели са предвидени с цел една и съща променлива-указател да може в различни моменти да сочи към данни от различен тип. В този случай, при опит да се използва съдържанието на променливата от тип указател, ще се предизвика грешка. Съдържанието на променлива-указател към тип **void** може да се извлече само след привеждане на типа на указателя (**void***) до типа на съдържанието. Това може да се осъществи чрез операторите за преобразуване на типове.

Пример:

```
int a = 100;
void* p;// дефинира указател към void
p = &a;// инициализира p
cout << *p;// грешка
cout << *((int*) p);// преобразува p в указател към int
// и тогава извлича съдържанието му.
```

В C++ е възможно да се дефинират указатели, които са константи, а също и указатели, които сочат към константи. И в двата случая се използва запазената дума const, която се поставя пред съответните елементи от дефинициите на указателите. Стойността на елемента, дефиниран като const (указателя или обекта, към който сочи) не може да бъде променяна.

Пример:

```
int i, j = 5;
int *pi;// pi е указател към int
int * const b = &i;// b е константен указател към int
const int *c = &j;// c е указател към цяла константа.
b = &j; //грешка, b е константен указател
*c = 15; // грешка, *c е константа
```

Указатели и масиви

В C++ има интересна и полезна връзка между указателите и масивите. Тя се състои в това, че имената на масивите са указатели към техните “първи” елементи. Това позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив.

Указатели и едномерни масиви

Нека **a** е масив, дефиниран по следния начин:

int a[100];

Тъй като **a** е указател към a[0], *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a+1 е адреса на a[1], a+2 е адреса на a[2] и т.н. a+n-1е адреса на a[n-1]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, ..., n-1).

```
#include <iostream>
using namespace std;
int main()
{ int n=15, a[3]={10,20,30};
  int* pn,k;
  pn=&n;
  cout <<" "<<pn<<" "<< *pn<<endl;
  cout <<" "<<a<<" a[0]="<< *a<<endl;
  cout <<" "<<a+2<<" a[2]="<<* (a+2)<<'\n';
  return 0;
}
```

Има обаче една особеност. *Имената на масивите са константни указатели. Заради това, някои от аритметичните операции, приложими над указатели, не могат да се приложат над масиви. Такива са ++, --и присвояването на стойност.*

Следващата програма показва два начина за извеждане на елементите на масив.

```
#include <iostream>
using namespace std;
int main()
{
 int a[] = {1, 2, 3, 4, 5, 6};
 for (int i = 0; i <= 5; i++)
   cout << a[i] << '\n';
 for (int i = 0; i <= 5; i++)
   cout << *(a+i) << '\n';
 return 0;
}
```

Фрагментът

```
for (int i = 0; i <= 5; i++)
 {cout << *a << '\n';
  a++;
  }
```

съобщава за грешка заради оператора a++ (a е константен указател и не може да бъде променян). Може да се поправи като се използва помощна променлива от тип указател към int, инициализирана с масива a, т.е.

```
int* p = a;
for (int i = 0; i <= 5; i++)
 {
  cout << *p << '\n';
  p++;
 }
```

Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се преобразуват в *(a+i), т.е. операторът за индексиране [...] се обработва от компилатора чрез адресна аритметика. Полезно е да отбележим, че операторът [] е лявоасоциативен и с по-висок приоритет от унарните оператори (в частност от оператора за извличане на съдържание *).

Указатели и низове

Низовете са масиви от символи. Името на променлива от тип низ е константен указател, както и името на всеки друг масив. Така всичко, което казахме за връзката между масив и указател е в сила и за низ–указател.

Следващият пример илюстрира обхождане на низ чрез използване на указател към char. Обхождането продължава до достигане на знака за край на низ.

```
#include <iostream>
using namespace std;
int main()
{
  char str[] = "C++Language";// str е константен указател
  char* pstr = str;
  while (*pstr)
   {
     cout << *pstr << '\n';
     pstr++;// pstr вече не е свързан с низа “C++Language”.
    }
  return 0;}
```

Тъй като низът е зададен чрез масива от символи str, str е константен указател и не може да бъде променяна стойността му. Затова се налага използването на помощната променлива pstr.

Ако низът е зададен чрез указател към char, както е в следващата програма, не се налага използването на такава.

```
#include <iostream>
using namespace std;
int main()
{
  char* str = "C++Language"; // str е променлива
  while (*str)
   {
    cout << *str << '\n';
    str++;
    }
  return 0;
}
```

Примерите показват, че задаването на низ като указател към char има предимство пред задаването като масив от символи. Ще отбележим обаче, че дефиницията

```
char* str = "C++Language";
```
не може да бъде заменена с
```
char* str;
cin >> str;
```
следвани с въвеждане на низа “C++Language”, докато дефиницията
```
char str[20];
```
позволява въвеждането му чрез cin, т.е.
```
cin >> str;
```
Има още една особеност при дефинирането на низ като указател към char. Ще я илюстрираме с пример.

Нека дефинираме променлива от тип низ по следния начин:

```
char s[]="asd";
*s='A';
cout<<s<<endl;
```
Операторът
```
*s='A';
```
е еквивалентен на `s[0]='A'` и ще замени първото срещане на символа `'a'`

в **s** с `'A'`. Така

```
cout<<s;
```

извежда низа

Asd

Да разгледаме съответната дефиниция на s чрез указател към char

```
int main ()
{
/* грешка
char* s="asd";
*s='A';
cout<<s<<endl;
*/
```
Операторът

```
*s='A';
```

би трябвало да замени първото срещане на символа `'a'` в s с `'A'`, тъй като s съдържа адреса на първото ‘a’. Тук обаче компилаторът съобщава за грешка – нарушение на достъпа.

Тип псевдоним

Чрез псевдонимите се задават алтернативни имена на обекти в общия смисъл на думата (променливи, константи и др.). В тази част ще ги разгледаме малко ограничено (псевдоними само за променливи).

Задаване на тип псевдоним

Нека T е име на тип. Типът T& е тип псевдоним на T. T се нарича базов тип на типа псевдоним.

Множество от стойности

Състои се от всички имена на дефинирани вече променливи от тип T.

Пример: Нека програмата съдържа следните дефиниции

```
int a, b = 5;
...
int x, y = 9, z = 8;
...
int& w=x;
```

Множеството от стойности на типа int& съдържа имената a, b, x, y, z. Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип псевдоним, се нарича променлива от този тип псевдоним.

<дефиниция_на_променлива_от_тип_псевдоним> ::=T&<var> = <defined_var_of_T>; |

T &<var> = <defined_var_of_T>;

където

T е име тип, а

<defined_var_of_T> е име на вече дефинирана променлива от тип T. Нарича се инициализатор.

Възможно е фрагментите

<var> = <defined_var_of_T> и
&<var> = <defined_var_of_T>
да се повтарят многократно. За разделител се използва символът запетая. Има обаче една особеност. Дефиницията

T& a = b, c = d;
е еквивалентна на
T& a = b;
T c = d;
Пример: Дефинициите

int a = 5;
int& syna = a;
double r = 1.85;
double &syn1 = r, &syn2 = r;
int& syn3 = a, syn4 = a;
определят syna и syn3 за псевдоними на a, syn1 и syn2 за псевдоними на r и syn4 за променлива от тип int.

Дефинициите задължително са с инициализация – променлива от същия тип като на базовия тип на типа псевдоним. Освен това, след инициализацията, променливата псевдоним не може да се променя като ѝ се присвоява нова променлива чрез повторна дефиниция. Затова тя е "най-константната" променлива, която може да съществува.

Пример:

```
...
int a = 5;
int &syn = a; // syn е псевдоним на a
int b = 10;
int& syn = b;// error, повторна дефиниция
...
```

Операции и вградени функции

Дефиницията на променлива от тип псевдоним свързва променливата-псевдоним с инициализатора и всички операции и вградени функции, които могат да се прилагат над инициализатора, могат да се прилагат и над псевдонима ѝ и обратно.

Примери:

```
1. int ii = 0;
   int& rr = ii;
   rr++;
   int* pp = &rr;
```

Резултатът от изпълнението на първите два оператора е следния:

```
  ii, rr
... 0 ...
```

Операторът rr++; не променя адреса на rr, а стойността на ii и тя от 0 става 1. В случая rr++ е еквивалентен на ii++. Адресът на rr е адреса на ii. Намира се чрез &rr. Чрез дефиницията

int* pp = &rr;

pp е определена като указател към int, инициализирана с адреса на rr.

```
2.  int a = 5;
    int &syn = a;
    cout << syn << " " << a << '\n';
    int b = 10;
    syn = b;
    syn++;
    cout << b << " " << a << " " << syn << '\n';
```

извежда

```
5 5
10 11 11
```

Операторът syn = b; е еквивалентен на a= b;.

```
3.  int i = 1;
    int& r = i; // r и i са свързани с едно и също цяло число
    cout << r; // извежда 1
    int x = r;// x има стойност 1
    r = 2;// еквивалентно е на i = 2;
```

Допълнение: Възможно е типът на инициализатора да е различен от този на псевдонима. В този случай се създава нова, наречена временна, променлива от типа на псевдонима, която се инициализира със зададената от инициализатора стойност, преобразувана до типа на псевдонима.

Например, след дефиницията

```
double x = 12.56;
int& synx = x;
```

имаме

x	synx
12.56...	12
8B	4B

Сега x и псевдонимът ѝ synx са различни променливи и промяната на x няма да влияе на synx и обратно.

Константни псевдоними

В C++ е възможно да се дефинират псевдоними, които са константи. За целта се използва запазената дума const, която се поставя пред дефиницията на променливата от тип псевдоним. По такъв начин псевдонимът не може да променя стойността си, но ако е псевдоним на променлива, промяната на стойността му може да стане чрез промяна на променливата.

Пример: Фрагментът

```
int i = 125;
const int& syni = i;
cout << i << " " << syni << '\n';
syni = 25;
cout << i << " " << syni << '\n';
```

ще съобщи за грешка (syni е константа и не може да е лява страна на оператор за присвояване), но фрагментът

```
int i = 125;
const int& syni = i;
cout << i << " " << syni << '\n';
i = i + 25;
cout << i << " " << syni << '\n';
```

ще изведе

```
125 125
150 150
```

Последното показва, че константен псевдоним на променлива защитава промяната на стойността на променливата чрез псевдонима.

Указателите и псевдонимите като параметри на функции

Задача 1**:** *Да се напише програма, която въвежда стойности на естествените числа a, b, c и d и намира и извежда най-големият им общ делител.*

Програмата по-долу решава задачата.

```
int gcd(int x, int y)
{
  while (x != y)
   if (x > y) x = x-y; else y = y-x;
  return x;
 }
 int main()
 {
  cout << "a, b, c, d= ";
  int a, b, c, d;
  cin >> a >> b >> c >> d;
  int r = gcd(a, b);
  int s = gcd(c, d);
  cout << "gcd{" << a << ", " << b << ", " << c << ", " << d
       << "}= " << gcd(r, s) << "\n";
   return 0;
  }
```

Тя се състои от две функции **:** **gcd** и **main**. Функцията **gcd**(x, y) намира най-големия общ делител на естествените числа x и y. Тъй като **main** се обръща към (извиква) **gcd**, функцията **gcd** трябва да бъде известна преди функцията **main**.

Функцията **gcd** реализира най-простото и "чисто" дефиниране и използване на функции – получава входните си стойности единствено чрез формалните си параметри и връща резултата от работата си чрез оператора return. Забелязваме, че обръщението **gcd**(a, b) работи с копие на стойностите на a и b, запомнени в x и y, а не със самите стойности на формалните параметри. В процеса на изпълнение на тялото на **gcd**, стойностите на x и y се променят, но това не оказва влияние на стойностите на фактическите параметри a и b.

Такова свързване на формалните с фактическите параметри се нарича **свързване по стойност** или още **предаване на *параметрите по стойност***. При него фактическите параметри могат да бъдат не само променливи, но и изрази от типове, съвместими с типовете на съответните формални параметри.

Пример:

```
#include <iostream>
using namespace std;
int F(int x, int y)
{
 int z;
 z=x*x-y*y;
 return z;}

int main()
```

```
{
 int a,b;
 cin>>a>>b;
 cout<<a<<" "<<b<<endl;
 cout<<F(a-b,a+b)<<endl;
 cout<<a<<" "<<b;
}
```
Ако се въведат стойности 6 3
се извеждат 6 3
-72
6 3

Предаване на параметри чрез указател

В този случай в стековата рамка на функцията за формалния параметър се отделят 4B, в които се записва стойността на фактическия параметър, която е адрес на променлива. Действията, описани в тялото се изпълняват със съдържанието на формалния параметър – указател. По такъв начин е възможна промяна на стойността на променливата, чийто адрес е предаден като фактически параметър.

В редица случаи се налага функцията да получи входа си чрез някои от формалните си параметри и да върне резултат не по обичайния начин – чрез оператора return, а чрез същите или други параметри.

Пример:
```
#include <iostream>
using namespace std;
void F(int *x, int *y)
{int z;
 z=*x;
*x=*y;
 *y=z;
 }
int main()
{int a,b;
    cin>>a>>b;
    cout<<a<<" "<<b<<endl;
    F(&a,&b);
    cout<<a<<" "<<b;
}
```
Ако се въведат стойности 4 8
се извеждат 4 8
8 4

Освен тези два начина на предаване на параметри, в езика C++ има още един – **предаване на *параметри по псевдоним***. Той е сравнително по-удобен от предаването по указател и се предпочита от програмистите.

Формалният параметър-псевдоним се свързва с адреса на фактическия. За него в стековата рамка на функцията памет не се отделя. Той просто “прелита” и се “закача” за фактическия си параметър. Действията с него се извършват над фактическия параметър.

Пример:

```
#include <iostream>
using namespace std;
int F(int &x, int &y)
{int z;
 z=x;
 x=y;
 y=z;
 return z;
}
int main()
{int a,b;
 cin>>a>>b;
 cout<<a<<" "<<b<<endl;
 cout<<F(a,b)<<endl;
 cout<<a<<" "<<b;
 }
```

Ако се въведат стойности 5 2
се извеждат 5 2
5
2 5

Структури от данни. Обектно-ориентираното програмиране. Библиотеката STL.

Структури от данни (СД) *са начин на организация на даден тип данни в паметта на компютъра обикновено с цел ефективност. Осигуряват удобство за добавяне и/или премахване на нови елементи. СД имат свойства, като например бърз достъп до първия/ последния/ и др. елементи. Казано по-точно СД е комбинацията от стойностите на различни данни, техните взаимовръзки и операции които могат да бъдат извършени с тези данни.*

СД е програмна единица, която позволява да се съхранява и обработва от еднотипни и/или логически свързани данни чрез компютър. По-точно всяка величина (константа или променлива), определена в програма се нарича структура от данни. Неформално казано, всяка СД е конкретен представител на съответен тип данни.

За да се определи една структура от данни е необходимо да се направи:

- логическо описание на структурата, което я описва на базата на декомпозицията ѝ на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции.

- физическо представяне на структурата, което дава методи за представяне на структурата в паметта на компютъра.

В Увод в програмирането разгледахме структурите числа, логически и символи. За всяка от тях в езика C++ са дадени съответни типове данни, които ги реализират.

Типове данни (ТД) *се характеризират с: множество от стойности и операции над стойностите.*

Тъй като елементите на тези структури се състоят от една компонента, те се наричат **прости**, или **скаларни**. Такъв е и тип указател, който ще разгледаме по-късно.

Структури от данни, компонентите на които са редици от елементи, се наричат **съставни**. В езика C++ структурата масив се реализира чрез типа масив.

Съставните типове данни се разделят на хомогенни или нехомогенни. Ако всички компоненти на съставна структура от данни са от един и същи тип, тогава структурата е хомогенна, а типът на компонентите ѝ определя нейния базов тип данни. В противен случай, т.е. ако компонентите на структурата са от различни типове, структурата е нехомогенна.

Структури от данни, за които операциите включване и изключване на елемент не са допустими, се наричат **статични**, в противен случай - **динамични**.

Всеки съставен ТД в частност може да се разглежда като СД.

Пример: int[] е ТД масив от елементи от тип int и може да се разглежда като СД, която представя редица от цели числа последователно в паметта

Компютрите могат да съхраняват и обработват огромни количества данни. Формалните структури от данни позволяват на програмиста да структурира мислено големи количества данни в концептуално управляеми връзки. Понякога използваме структури от данни, за да можем да направим повече: например за бързо търсене или сортиране на данни. Друг път използваме структури от данни, за да можем да направим по-малко: например, концепцията за стека е ограничена форма на по-обща структура от данни. Тези ограничения ни предоставят гаранции, които ни позволяват да разсъждаваме по-лесно за нашите програми. Структурите от данни също така осигуряват гаранции за алгоритмичната сложност - изборът на подходяща структура от данни за работа е от решаващо значение за писането на добър софтуер.

Тъй като структурите от данни са абстракции от по-високо ниво, те ни представят операции с групи данни, като добавяне на елемент в списък или търсене на елемент в приоритетна опашка. Когато структурата от данни осигурява операции, можем да наречем структурата от данни абстрактен тип данни (понякога съкратено като ADT). Абстрактните типове данни могат да минимизират зависимостите в кода, което е важно, когато кодът трябва да бъде променен. Тъй като сме се абстрахирали от детайли от по-ниско ниво, някои от общите черти на по-високо ниво, които една структура от данни споделя с друга структура от данни, могат да бъдат използвани за замяна на едната с другата.

Всяка структура от данни може да се разглежда като една единица, която има набор от стойности и набор от операции, които могат да бъдат извършени за достъп или промяна на тези стойности. Самата структура от данни може да се разбира като набор

от операции над структурата от данни заедно със свойствата на всяка операция (т.е. какво прави операцията и колко време ще отнеме и т.н).

Езиците за програмиране притежават набор от вградени типове, като например цели числа и числа с плаваща запетая, които ни позволяват да работим с обекти от данни, за които процесорът на машината има вградена поддръжка. Тези вградени типове са абстракции от това, което всъщност предоставя процесорът, защото вградените типове скриват подробностите както за изпълнението, така и за ограниченията.

Обектно-ориентираното програмиране

При обектно-ориентираното програмиране водеща е парадигмата, че данните могат да описват определени същности, които можем да срещнем и в реалния живот. Тук идват обектите. Обектът описва поведението и характеристиките, които тук се наричат „методи“ и „свойства“. Освен това, за да бъде един език обектно-ориентиран не е достатъчно той да позволява работа с класове и обекти. ООП включва и други концепции като наследяване, капсулация, полиморфизъм и други. Нека разгледаме посочените концепции малко по-подробно.

При него водеща е парадигмата, че данните могат да описват определени същности, които можем да срещнем и в реалния живот. Тук идват обектите. Обектът описва поведението и характеристиките, които тук се наричат „методи“ и „свойства“. Освен това, за да бъде един език обектно-ориентиран не е достатъчно той да позволява работа с класове и обекти. ООП включва и други концепции като наследяване, капсулация, полиморфизъм и други. Нека разгледаме посочените концепции малко по-подробно.

Капсулация

Капсулацията (известна още и като „information hiding, „скриване на информация“) е един от основните принципи на обектно-ориентираното програмиране. При нея един обект трябва да предоставя на ползващия го само пряко-необходимите му средства за управление. Например един офис служител използва единствено периферията на едно PC. Той няма нужда да познава модела на процесора, неговата архитектура, начина на връзка с дънната платка и т.н. Тези компоненти и знанията за тях (процесор, дънна платка) се явяват скрити за него. Този, който създава класовете определя какво да е скрито и какво да е публично видимо. Това става чрез изрично дефиниране като private (скрит) всяко поле или метод, които не искаме да се ползват от друг клас.

Наследяване

Наследяването също е основен принцип в ООП. Чрез него един клас може да „наследява“ методи и свойства от друг, по-общ клас. Позволете ми да приведа метафоричен пример с животни. Разред Хищници се характеризира с: четириноги, бозайници, ядат месо. Вълкът е от семейство „Canidae” (което се превежда „кучета“).

Всички вълци са четириноги, бозайници, ядат месо. Тези характеристики на вълка могат да се зададат на някой по-общ клас (в случая разред хищници) и към този по-висок клас да се свързват всички отговарящи на тази характеристика класове т.е. те черпят тази обща информация от този по-висок клас, което ни спестява да я изписваме всеки път, за всеки клас който искаме да има тези качества. Този клас, който наследяваме се нарича клас-родител или още базов клас (base class, super class).

Полиморфизъм

Още един от важните принцип на ООП е т.нар. „полиморфизъм". Нека се върнем на примера с животните. Дефинирали сме даден клас (семейство „Canidae"). Да, само че в това „семейство" има много и разнообразни хищници – от вълци, през лисици и стигайки до всички кучета. Те имат различно поведение по отношение на лова. Полиморфизмът ни позволява да третираме кое да е от тези животни като член на семейство „Canidae" и да му зададем команда да извърши улов, без значение какво точно е „животното", на което задаваме командата. За да извърши това полиморфизмът използва пренаписване на методи в наследените класове, с цел промяна на първоначалното им поведение, което е прихванато от базовия клас.

Тези, и други принципи залегнали в ООП спомагат за реализирането на неговата цел – да позволи по-бърза разработка на сложен софтуер, както и по-лесната му поддръжка след това, както и интуитивното преизползване на кода в други приложения.

Класове и обекти

Простите типове (целочислен, реален и символен) затрудняват програмистите, когато искат да представят в програмите си различни съвкупности от данни като нещо цяло и единно. Едно ограничено улеснение в това отношение са масивите. Масивът е съставен тип данни, който позволява определена съвкупност от данни да се представи в програмата като един масив. При това програмистът сам конструира масива, т.е. сам взема решение едномерен или многомерен масив да използва в своята програма. За обработката на масиви обикновено се създават цикли, с което програмите стават кратки и прегледни. Масивите, обаче са подчинени на едно важно условие – всичките елементи на даден масив трябва да са от един и същи тип. Езикът C++ предлага средство за създаване на съвкупности от различни по тип компоненти. Такова средство са класовете.

Класовете в C ++ са типове, състоящи се от множество компоненти, които могат да бъдат данни и функции. Данните принадлежащи на даден клас, се наричат данни членове на този клас или член-данни, а функциите, принадлежащи на класа - функции членове на класа член-функции. *Класовете в една програма се създават от програмиста в съответствие на конкретните нужди на програмата. Член-данните могат да бъдат от познатите прости типове (целочислени, реални и символни), масиви или от друг, вече създаден клас.*

След като е създаден даден клас, могат да се дефинират променливи от този клас, наречени обекти. Връзката между клас и обект е такава, каквато е връзката между тип и променлива. Но обектите не са обикновени променливи, тъй като се състоят от множество компоненти, включително и функции. Обектите са структурните единици на обектно-ориентираните програми, чрез които се моделират различни реални процеси и обекти.

Дефиниране на класове

Дефиницията на клас представлява описание на нов тип, който се състои от множество компоненти. Синтаксисът на дефиницията на клас е следният

class [Име_на_класа] { Описание на компонентите на класа };

декларация – описание на шаблона на класа:

```
class име_на_класа
{
режим_на_достъп:
        декларация_на_компоненти;

режим_на_достъп:
        декларация_на_компоненти;
        ...
}; //декларацията завършва с ;
```

Методите се декларират само със своите прототипи в хедър файлове (обикновено) .h.

дефиниция на методите на класа (.cpp – файл)

```
тип_на_резултата име_на_класа_ :: име_на_метода (формални_параметри)
{
        ...
}
```

:: - параметър за принадлежност

Както се вижда, член-функциите на класа са функции, но когато не са вградени, имената им се състоят от две части: име на класа, на който принадлежи член-функцията, и име на самата член-функция, разделени чрез оператора за принадлежност ::. *Така образуваните имена на член-функциите се наричат пълни имена.*

Имената на класовете (имената, които се задават след ключовата дума class) трябва да бъдат уникални в рамките на една програма.

Компонентите на класовете могат да бъдат данни и функции. Имената им са локални, т.е. в различните класове може да има компоненти с еднакви имена.

Член-функциите могат да бъдат представени в класа по два начина:

- в класа присъства само прототипа на член-функцията, т.е. член-функцията само се декларира в класа, а нейната дефиниция е извън класа;

- в класа присъства цялата дефиниция на член-функцията – в този случай член-функцията е вградена функция, т.е. тя е inline.

Режим за достъп до компоненти на даден клас

Чрез режимите на достъп се постига скриване на член-данни на класа, това са член-данните дефинирани в скритата част на класа, това скриване на данни се нарича ***капсулиране***. ***То позволява данните и функциите, които действат върху един обект да останат скрити вътре в него. Достъпът до тях може да се контролира от програмите извличащи или модифициращи данните само чрез интерфейса на обекта.***

Достъпът на външни функции до компонентите на класовете се определя от начина на деклариране на компонентите. Всяка компонента на даден клас може да бъде декларирана като ***частна*** *(private:),* ***обща*** *(public:) или* ***защитена*** *(protected:). Компонентите на класовете са частни по подразбиране,* т.е. ако не е указана ключовата дума (private:,public: или protected:), се подразбира private:. *Частните компоненти (компонентите, които са декларирани след ключовата дума private:) не са достъпни за външни функции. Общите компоненти (компоненти, които са декларирани след ключовата дума public:) са достъпни за външни функции. Защитените компонентите (компоненти, които са декларирани след ключовата дума protected:) не са достъпни за външни функции, те се различават по достъп от частните при производните класове, които ще разгледани в следващите теми.*

Член-данните на класа Student са декларирани като частни, защото са след ключовата дума private:. Поради това единствено член-функциите getData() и display() на класа Student имат достъп до тях. Ако се направи опит за достъп отвън до компонентите *член-данните* на обектите A или B, компилаторът ще регистрира грешка. Чрез използването на частни компоненти се постига скриване на компоненти на обектите от външната среда (използва се още и терминът "**капсулиран**е").

Компонентите на даден клас, които трябва да бъдат видими извън класа (да бъдат достъпни за външни функции), трябва да бъдат декларирани като общи (public:).

Конструктури и деструктури

Създаването на обекти в редица случаи е свързано с определени действия, които се определят като **инициализация.** Най-често инициализацията се свежда до задаване на начални стойности на данните на новосъздадените обекти. В общия случай обаче, инициализацията може да включва и други действия като заделяне на памет, запомняне на текущото състояние на програмата с цел неговото възстановяване в друг момент и пр. От друга страна, унищожаването на обектите също може да бъде

свързано с определени действия, които се определят като **заключителни**. Заключителните действия най-често са свързани с освобождаване на преди това заделена динамична памет, възстановяване на предварително запомнено състояние на програмата и др. Очевидно е, че би било много удобно инициализацията и заключителните действия да се извършват автоматично при създаването на обектите и съответно при тяхното унищожаване. *Езикът C++ предоставя такава възможност, която се реализира чрез специален вид член-функции на класовете, наречени* ***конструктори и деструктори.***

Конструктор – метод, чието име съвпада с името на класа. Основното му предназначение е инициализация на данните на новосъздадения обект. Конструкторите се задействат автоматично, когато се създава обект от съответен клас.

```
class Temp{
   Temp();
   Temp(int, float);
   Temp(float, int);
   ...
};

Temp p, q(3, 4.25), r(7.06, 10);
```

В този пример конструкторите са три и се задействат автоматично при създаване на обектите p, q, r.

Деструктори – метод, чието име се образува от името на класа, предшествано от ~(~Temp). Основното му предназначение е нулиране на данни или унищожаване на динамично заделена памет. Деструкторите се изпълняват автоматично при унищожаване на динамични обекти или при завършване на функция, в която обектът е бил дефиниран.

```
#include <iostream>
#include <string.h>
#include <stdlib.h>
using namespace std;
class student
{  private:
          string ime;
          int ocenki[6];
          double s1;
   public:
          student() {ime=""; for (int i=0;i<6;i++)
                       ocenki[i]=0; s1=0; };
          student(string s, int a[6], double b)
             {ime=s;
              for (int i=0;i<6;i++)
                ocenki[i]=a[i]; s1=b;
             };
          void read();
          void print_st();
```

```
        string access_ime (){return ime;}
        double access_sr () {return s1;}
        int*   access_ocenki() {return ocenki;}
      };

void  student:: read()
{cout<<"Въведи нов студент:"<<endl;
  cout<<"Име:";cin>>ime;
  double s=0;
  cout<<"Въведи оценки по 6 дисциплини:";
  for(int j=0;j<6;j++){cin>>ocenki[j]; s+=ocenki[j];}
   s1=s/6 ;
 }

 void  student:: print_st()
{
  cout<<"Име:"<<ime<<" ";
  cout<<"Оценки :";
  for(int j=0;j<6;j++)
   {
     cout<<ocenki[j]<<" ";
     }
   cout<<"Среден успех :"<<s1<<endl;
}
int main()
{
  int z;
  student o1;   int temp[6]={4,4,4,4,4,4};
  student o2("Ivan", temp,6) ;
  o1.print_st();o2.print_st();
```

Библиотеката STL

Почти всички съвременни компилатори на C++ съдържат библиотеката STL. Първоначално е конфигурирана да работи с почти всякакви данни, което се осигурява от набор от шаблони за функции и класове.

Какво съдържа

Библиотека от шаблони, реализираща стандартни структури от данни и алгоритми.

- част от C++ Standard Library
- основни компоненти

- алгоритми (<algorithm>) Основни алгоритми: сортиране (sort()), обръщане (reverse()), размяна (swap()), минимум (min()), максимум (max()), премахване на дубликати (unique()), произволно разбъркване (random_shuffle()), следваща пермутация (next_permutation()), намиране на *n*-ти елемент (nth_element()).

- контейнери - основни структури данни: динамичен масив (vector), опашка (queue и deque), свързан списък (list), стек (stack), асоциативен масив (map и multimap),

множество (set и multiset), приоритетна опашка (priority_queue или heap), и (не точно в STL, но все пак) стринг (string).

- функционални обекти (<functional>).

Функционалните обекти са всички аритметични оператори: събиране (плюс), изваждане (събиране), умножение (пъти), деление (дели), вземане на остатъка (модул) и обръщане на знака (отрицание). Има функционални обекти за изчисляване на равенство (равно_то), неравенство (не_равно_то), операция "по-голямо" (по-голямо), операция "по-малко" (по-малко), операция "по-голямо или равно" (по-голямо_еднакво), операция "по-малко или равно" (по-малко_ равно) ... Логическите оператори имат свои собствени функционални обекти: логически "и" (логически_и), логически "или" (логически_или) и логически "не" (логически_не).

- базирана на идеята за обобщеното програмиране, т.е. програмиране, при което като параметри се използват типове.

Механизмът на шаблоните в C++ позволява използването на типове в качеството на параметри при дефинирането на функции и класове. Шаблонът зависи само от тези свойства на параметъра-тип, които явно използва. Поради това не е необходимо различните типове, които се използват като параметри на шаблона да бъдат свързани по какъвто и да било начин.

- ефективност: дава гаранции за сложност на алгоритми и операции над СД.

За "структури данни" в STL се ползва името *контейнери*. Това име идва поради специфичността на структурите от данни, които са реализирани в STL - това са само такива структури данни, които съдържат елементи (подредени по някакъв начин) и дават достъп до тях. Те рядко, ако изобщо, предоставят методи, които да връщат някаква по-специфична информация, както например тяхна сума, работа върху подинтервали или други подобни.

Ще разгледаме най-често ползваните от тях. Като цяло интерфейсът им е доста сходен (голяма част от методите се повтарят), в следствие на което нещото, което *реално* трябва да се запомни е *разликите* между структурите данни.

Трябва да се отбележи, че всички алгоритми работят с методите на контейнери, без да навлизат в подробностите за тяхната реализация. Така че, ако алгоритъмът трябва да определи дали един елемент от контейнера е равен на друг, той просто извиква предефинирания оператор за сравнение "operator = = ()", реализиран в контейнера.

Динамичен масив(Dynamic Array (Vector))

Най–често срещаните и използвани са линейните структури. Те представляват абстракция на всякакви видове редици, последователности, поредици и други подобни от реалния свят.

Понякога няма да знаем колко елемента най-много може да въведе потребителят, или ще искаме да ползваме възможно най-малко памет, за да ги запазим. Това обаче, е трудно за реализация, тъй като трябва да се занимаваме с динамично заделяне на памет и да се грижим тя да бъде освободена, след като сме спрели да я ползваме. Тъй като това е от една страна досадно, от друга все пак много често срещано, вместо това ще направим структура данни, която да прави това вместо нас.

Динамичен масив

Досега използвахме масиви, които по време на компилация запазват място в паметта и тази памет е ангажирана до края на програмата (независимо дали масивът ни е нужен още). Тези масиви са статични.

Видове заделяне на памет

По време на компилация (статично заделяне)

По време на изпълнение (динамично заделяне) – създадените променливи, обекти, класове и др. се наричат динамични.

Динамичен е масив, за който паметта се заделя по време на изпълнение. Той има същите свойства като статичния масив, но ползва само толкова памет, колкото му е нужна. За удобство, той обикновено пази и точно колко елемента съдържа, тъй като това е друго нещо, което почти винаги имаме и при статичните масиви. В литературата на английски най-често ще го срещнете като "vector" или "dynamic array".

Vector

#include <vector>

В библиотеката STL vector е динамичен масив с произволен достъп, най-често използван в случаите, когато трябва последователно да се добавят данни в края на редицата;

Забележка: Когато можете да ползвате обикновен масив - ползвайте обикновен масив. Понякога програма, която ползва вектори, може да е няколко пъти по-бавна от аналогична програма със стандартни масиви.

Често без STL бихме ползвали повече памет, но какво от това? Ако програмата се събира в ограниченията по памет, то няма проблем.

Контейнерът vector – съвкупност от данни от един и същ тип. До всеки елемент от тази съвкупност може да се осъществи индивидуален достъп. Основната му разликата от масив е динамичната му дължина, но не надминаваща 1073741823. Необходима е библиотека vector.

а. синтаксис:

vector<тип_на_стойностите> <име_на_вектора>[<първоначална_размерност>];

vector <int> t[3];
t.back()

б. вградени методи:

метод	семантика
[i]	Стойността на елемента на i-та позиция
back()	Връща стойността на последния елемент във вектора
begin()	Връща адреса на първата стойност на вектора
capacity()	Броя на елементите във вектора
clear()	Изтрива елементите във вектора
empty()	Проверява дали векторът съдържа елементи
end()	Връща адреса на последната стойност на вектора
erase(from, to)	Изтрива елементите във вектора от позиция from до позиция to в него
front()	Връща стойността на първия елемент
insert(val, to)	Вмъква елемент val, в зададена позиция to
max_size()	Връща максималната дължина на вектор
pop_back()	Изтрива последния елемент и намалява дължината с единица на вектора
push_back()	Добавя стойност в края на вектора
resize(new_size, val)	Преоразмеряване на вектор с new_size дължина и запълва новите елементи с val стойност
size()	Брой на елементите във вектора
swap(v)	Разменя стойностите в два вектора

<string> е специфична употреба на vector, където типът е предварително зададен (`char`).

Шаблонът за клас vector дефинира пълно множество от оператори в това число и оператора за сравняване. Една програма може да определи дали два вектора са равни и кой е по-голям или по малък от друг. За равни вектори се смятат 2 вектора с равен брой и еднакви елементи. Ето примерна програма, реализираща операции с вектори

```
#include <iostream>
#include<vector>
using namespace std;
vector<int> a (10, -1); // 10 ints with value -1
vector<int> b (20);     // 20 ints

void print()
{
 for(int i = 0;i < (int)a.size();i++)
  cout<< a[i]<<" ";
  cout<< "\n";
 }
 int main()
 {
  for(int i = 0;i < 20;i++)
     b[i] = i;
     print(); // -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
//~~~~ ASSIGN ~~~
//Присвоява ново съдържание на вектора, като отпадат всички елементи,
//съдържащи се във вектора преди извикването и ги замества с
//тези, посочени от параметрите:
 a.assign (5, 0);  // a repetition 5 times of value 0
 print();  // 0 0 0 0 0
 int arr[] = {11, 21, 31, 41, 51, 61, 71};
 a.assign (arr + 2, arr + 6);// присвояване от масив.
 print(); //31 41 51 61
 a.assign (b.begin() + 5, b.end() -10);// присвояване от друг вектор
 print(); // 5 6 7 8 9

 //~~~~ BACK ~~~
 //Указател към последния елемент във вектора.
 while(a.back() != 0)
 {a.push_back(a.back() -1);}
 print(); //5 6 7 8 9 8 7 6 5 4 3 2 1 0
 //~~~~ BEGIN ~~~
 //Връща указател към първия елемент във вектора.
 vector<int>::iterator
 begin = a.begin();
 cout<<*begin<<endl; // 5
 //~~~~ END ~~~
 //Връща указател след последния елемент
 for(; begin < a.end(); begin++)
 cout<< *begin<<" " ; //5 6 7 8 9 8 7 6 5 4 3 2 1 0
 //~~~~ CAPACITY ~~~
 // Връща размера на разпределения капацитет за съхранение
 cout<<"\ncapacity: "<<a.capacity();//capacity: 20
 //~~~~ CLEAR ~~~
 //Изчиства съдържанието
 //Всички елементи на вектора се изтриват, като векторът
 // остава с размер 0.
   a.clear();print(); //
  //~~~~ EMPTY ~~~
  //Проверява дали векторът  е празен, т.е. размерът му е 0.
  cout<<(a.empty() ? "Empty.\n": "Not empty\n")<<endl; //Empty
  for(int i = 0;i < 10;i++)
  a.push_back(i + 1);
  print(); // 1 2 3 4 5 6 7 8 9 10
  //~~~~ ERASE ~~~
  //Премахва от вектора или отделен елемент (позиция), или набор от
  //елементи ([първи, последен)).
  // изтрива  6-тия елемент
  a.erase (a.begin() + 5);
  print(); // 1 2 3 4 5 7 8 9 10
  // изтрива първите 3 елемента:
  a.erase (a.begin(), a.begin() + 3);
  print(); // 4 5 7 8 9 10
 return 0;
 }
```

Чрез указатели са реализирани всички функции с различните структури от данни. Ето пример за възможна реализация на функциите push_back() и pop_back() за структурата vector.

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

```
using namespace std;
 int* a;int n=0;

 void push_back(int q)
  {int* pom=a;
   a=new int[n+1]; //нов динамичен масив с 1 елемент по дълъг
   //прехвърляне на елементите в новия масив
   for (int i=0;i<n;i++)
    { a[i]=pom[i];
     }
   a[n]=q;   n=n+1;
   delete []pom;//Изтриваме помощния масив
   }

  void pop_back()
  {int* pom=a;
    n=n-1;a=new int[n]; //нов динамичен масив с 1 елемент по къс
   //прехвърляне на елементите в новия масив без последния
    for (int i=0;i<n;i++)
      {a[i]=pom[i];
               }
   delete []pom;//Изтриваме помощния масив
   }

   void print()
    {for(int i=0;i<n;i++)
      cout<<a[i]<<" ";
      cout<<endl;
     }

  int main()
  { system("chcp 1251");//за кирилизиране на изхода
    int w, z;
    do
    {
     cout<<"Меню "<<endl;
     cout<<"0. за край!!"<<endl;
     cout<<"1. Добавяне на нов елемент в края"<<endl;
     cout<<"2. Изтриване на последния"<<endl;
     cout<<"3. Отпечатване"<<endl;
     cout<<"Избери от 0 до 3"<<endl;
     cin>>z;
     switch (z)
      {
       case 1: cin>>w; push_back(w);break;
       case 2:  pop_back();break;
       case 3: print();break;
       }
     }
    while (z!=0);
    }
```

Задача 1. Да се напише програма, която прочита текст, до въвеждане на нов ред, премахва повтарящите се думи и го отпечатва на екрана.

Вход:

aa ab ac aa aa ba ab cc
Изход:
aa ab ac ba cc

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

int main()
{
     vector<string> v;
     string s;
     char ch;
     int i, size;

     do
     {
       cin.get(ch);
       while(ch!=' '&& ch!='\n')
        {
          s.push_back(ch);
          cin.get(ch);
        }
          size = v.size();
          for(i = 0; i < size; i++)
            if(v[i] == s) break;
          if(i == size) v.push_back(s);
          s.clear();
      }
     while(ch != '\n');

     size = v.size();
     for(i = 0; i < size; i++)
       cout << v[i] << ' ';
  }
```

Задача 2. Да се напише програма, която прочита думи до @ и създава честотен речник на думите

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

int main()
{
   string s;
```

```
    vector<string> words;
    vector<int> count;
    int n, i;
    cin >> s;
    while(s!="@")
    {
         n = words.size();
         for(i = 0; i < n; i++)
              if(words[i] == s)
              {
                   count[i]++;
                   break;
              }
         if(i == n)
         {
              words.push_back(s);
              count.push_back(1);
         }
         cin >> s;
    }

    n = words.size();
    for(i = 0; i < n; i++)
    {
         int x = count[i];
         s = words[i];
         cout << s <<" "<<x<< endl;
     }

}
```

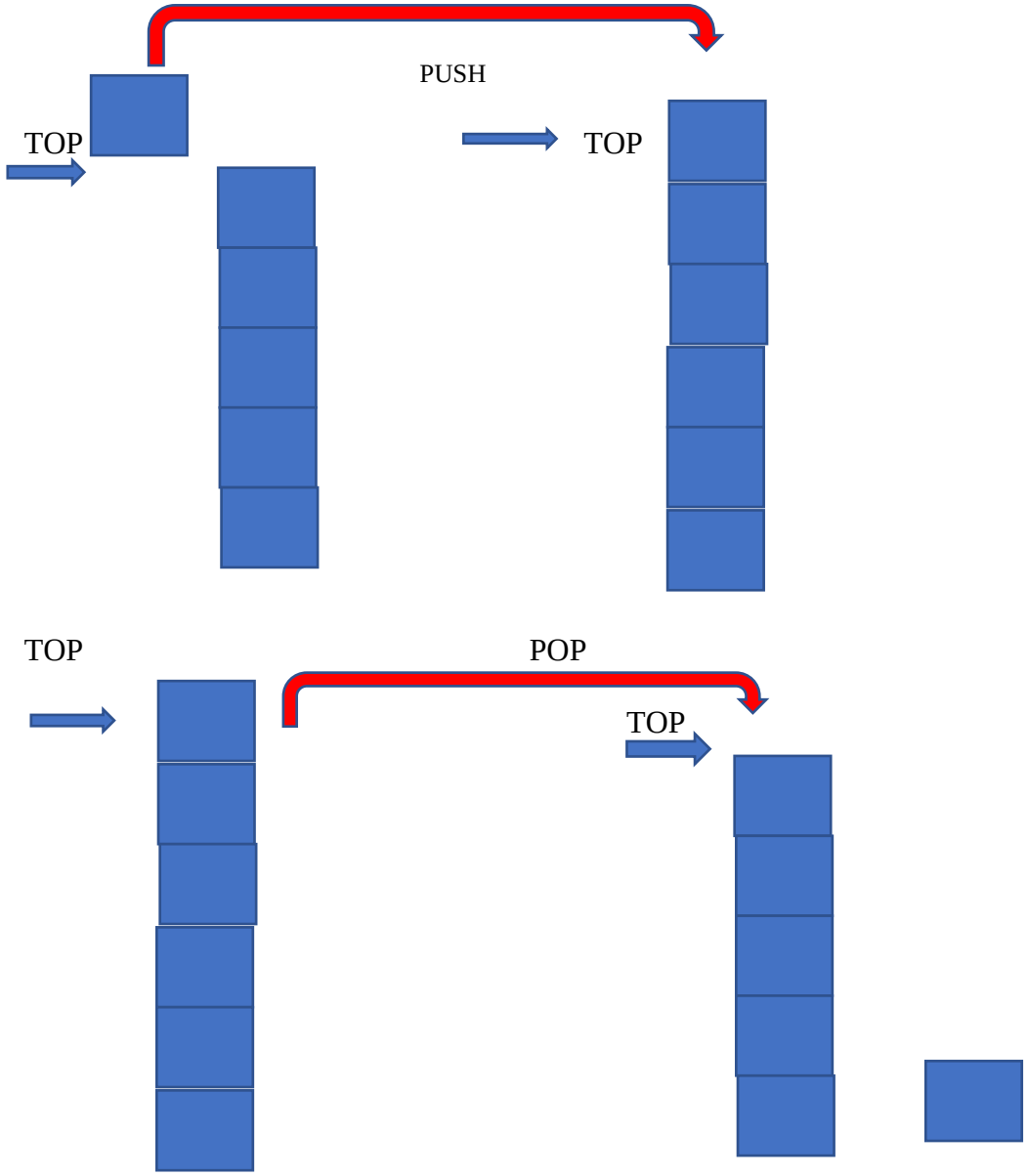
Стек

Стек или Stack на английски е третата от най-базовите абстрактни структури, които ще срещате отново и отново (другите две са Списък и Опашка).

Стекът е основна структура от данни, която логично може да се разглежда като линейна структура, представена от реален физически стек или купчина, структура, при която вмъкването и изтриването на елементи се извършва в единия край, наречен връх на стека. Стекът е динамична структура от данни с променлив брой елементи от един и същ тип.

Основната концепция може да бъде илюстрирана, като мислите за вашия набор от данни като купчина чинии или книги, където можете да свалите само горния елемент от стека, за да премахнете нещата от него. Тази структура се използва много в програмирането.

Основната реализация на стека се нарича още LIFO (Last In First Out), за да се демонстрира начинът, по който осъществява достъп до данните, тъй като има различни варианти на реализация на стека.



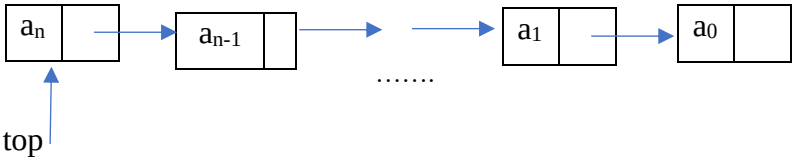
Има три основни операции, които могат да се извършват върху стека. Те са

1) вмъкване на елемент в стека (push)

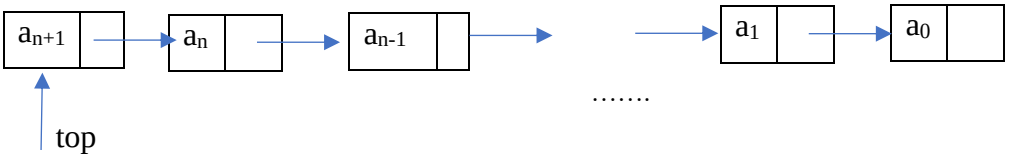
2) изтриване на елемент от стека (pop).

3) показване на съдържанието на върха на стека (top).

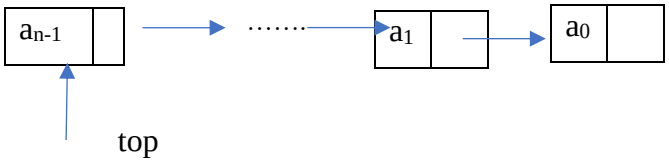
Стандартната имплементация е реализирана на базата на deque. Реализацията чрез свързан списък е една от най-лесните реализации на стека.



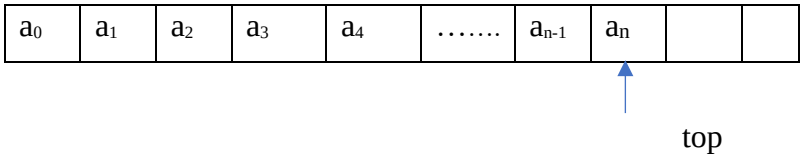
- вмъкване на елемент в стека



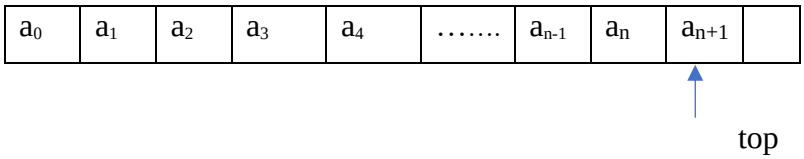
- изтриване на елемент от стека



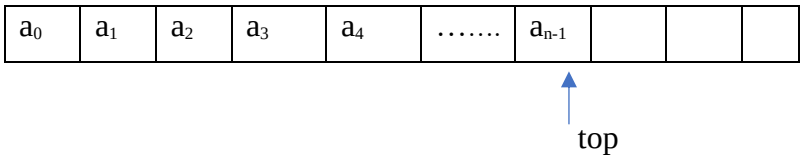
Обикновено дължината на стека се счита за неограничена. Ако дължината му се фиксира може да се реализира с масив.



- вмъкване на елемент в стека



- изтриване на елемент от стека



За да се позволи неограничен растеж (за сметка на известно забавяне) има по-голям смисъл да се прилага Vector.

Stack в STL

#include <stack>

Поддържа стандартните операции за стек.

- .top() - връща елемента на върха на стека без да го премахва.
- .push(T val) - добавя нов елемент със стойност val на върха на стека.
- .pop() - премахва елемента, намиращ се на върха на стека.
- .size() - връща колко елемента има в стека.
- .empty() - връща true или false в зависимост дали стекът е празен или не.

Възможен проблем е, стекът да се препълни (познато като stack overflow). За това се говори в темата за Рекурсия. Всъщност езиците за програмиране реализират рекурсия именно използвайки стек за менажиране на паметта, съответно проблемът при нея идва от проблема на стека, а не обратното.

Приложения на стека

Тази структура има широко приложение в компютрите - тя е основна част от изпълнението на програмите (програмния стек), както и участва в различни алгоритми. Стекът се ползва в един от най-популярните алгоритми в графи - Търсене в Дълбочина. Употребата му там, обаче, в повечето случаи е невидима за нас, тъй като се ползва рекурсия. В редица други алгоритми, обаче, се налага да се ползва директно стек. Понякога, когато задачата има специфична структура на решенията, стек се ползва и за оптимизация на Динамично Оптимиране.

Стекът има още приложения, някои от които са изброени по-долу.

1.*Преобразуване на десетично число в двоично число*

Задача 1. Да се напише програма, която въвежда десетично число n и извежда двоичното му представяне.

Логиката за превръщане на десетично число в двоично число е следната:

* Прочетете число
* Итерация (докато числото е по-голямо от нула)
1. Открийте остатъка след разделяне на числото на 2
2. Отпечатайте остатъка
3. Разделете числото на 2
* Край на итерацията

13(10) = 11012 1.23+1.22+0.2+1.1

Има обаче проблем с тази логика. Да предположим, че числото, чиято двоична форма искаме да намерим, е 23. Използвайки тази логика, получаваме резултата като 11101, вместо да получаваме 10111.

За да разрешим този проблем, използваме стек. Използваме свойството LIFO на стека. Първоначално поставяме двоичната цифра в стека, вместо да я отпечатваме

директно. След като цялото число е преобразувано в двоична форма, изваждаме една по една цифрите от стека и ги отпечатваме. Така получаваме десетичното число, преобразувано в правилната му двоична форма.

Алгоритъм:

1. Създайте стек
2. Въведете десетично **число**, което трябва да бъде преобразувано в еквивалентната му двоична форма.
3. Итерация1 (докато **число**то> 0)
3.1 **цифра** = **число**% 2
3.2 Вмъкнете **цифра**та в стека
3.3 Разделете **число**то на 2
4. Край на итерация1
5.iteration2 (докато стекът не е празен)
5.1 Взимаме **цифра** от стека
5.2 Отпечатваме **цифр**ата
6. Край на итерация2
7. СТОП

```
#include<stack>
#include<iostream>
using namespace std;
int main()
{ stack <int> d;
  int n;
  cin>>n;

 while (n !=0)
 {
   d.push(n%2);
   n=n/2;
  }
  while(!(d.empty()))
   {
     cout<<d.top() ;
     d.pop();
   }
   return 0;
   }
```

2. Пресмятане на израз и синтактичен анализ

Стекът се използва най-често при обработката на аритметичните изрази при транслация, които се преобразуват в обратен полски запис. Ако знакът на всяка операция стои след операндите, полския запис е обратен и се нарича постфиксен. Ако знакът предшества операндите полския запис е префиксен. Записът, при който знаците са между операндите (както в математиката се нарича инфиксен). Преобразуването от една форма на израза в друга форма може да се извърши с помощта на стек. Привеждането в полски запис се осъществява с премахване на скобите.

Много компилатори използват стек за анализиране на синтаксиса на изрази, програмни блокове и т.н. преди да се преведе в код на ниско ниво. Повечето от езиците за програмиране са езици без контекст, което им позволява да бъдат анализирани с машини, базирани на стека.

2.1 *Преобразуване на инфиксен израз, заграден в кръгли скоби, в посфиксен израз*

Input: (((8 + 1) - (7 - 4)) / (11 - 9))

Output: 8 1 + 7 4 - - 11 9 - /

Анализ: Има пет вида входни знаци, които са:

* Отварящи скоби
* Числа
* Оператори
* Затварящи скоби
* Символ от нов ред (\ n)

Изискване: стек от символи

Алгоритъм:

1. Прочетете символ
2. Действия, които трябва да се извършат след всеки вход

Отваряща скоба	(2.1) Включете го в стека и след това преминете към стъпка (1)
Число	(2.2) Отпечатайте го и след това преминете към стъпка (1)
Оператор	(2.3) Включете го в стека и след това преминете към стъпка (1)
Затваряща скоба	(2.4) Извадете елемента от върха на стека
	(2.4.1) Ако е оператор, отпечатайте го, преминете към стъпка (1)
	(2.4.2) Ако изскачащият елемент е отваряща скоба, извадете го и преминете към стъпка (1)
Символ от нов ред	(2.5) СТОП

Следователно крайният изход при преобразуване от инфиксен в посфиксен запис е както следва:

Вход	**Операция**	**Стек (след операцията**	**Изведи на екрана**
(	(2.1) Постави в стека	(	
(	(2.1) Постави в стека	((	
(	(2.1) Постави в стека	(((	
8	(2.2) Изведи го		8
+	(2.3) Постави в стека	(((+	8
1	(2.2) Изведи го		8 1
)	(2.4) Извади го от стека и след това изведи '+' на екрана	(((	8 1 +
	(2.4) Извади (от стека	((	8 1 +

Вход	Операция	Стек (след операцията	Изведи на екрана
-	(2.3) Постави в стека	((-	
(	(2.1) Постави в стека	((- (	
7	(2.2) Изведи го		8 1 + 7
-	(2.3) Постави в стека	((- (-	
4	(2.2) Изведи го		8 1 + 7 4
)	(2.4) Извади го от стека и след това изведи '-' на екрана	((- (	8 1 + 7 4 -
	(2.4) Извади (от стека	((-	
)	(2.4) Извади го от стека и след това изведи '-' на екрана	((	8 1 + 7 4 - -
	(2.4) Извади (от стека	(	
/	(2.3) Постави в стека	(/	
(	(2.1) Постави в стека	(/ (	
11	(2.2) Изведи го		8 1 + 7 4 - - 11
-	(2.3) Постави в стека	(/ (-	
9	(2.2) Изведи го		8 1 + 7 4 - - 11 9
)	(2.4) Извади го от стека и след това изведи '-' на екрана	(/ (	8 1 + 7 4 - - 11 9 -
	(2.4) Извади (от стека	(/	
)	(2.4) Извади го от стека и след това изведи '/' на екрана	(	8 1 + 7 4 - - 11 9 - /
	(2.4 Извади (от стека	Стекът е празен	
Нов ред	(2.5) Стоп		

```
#include <iostream>
#include <stdlib.h>
#include <stack>
using namespace std;
int main()
{
  stack <char> d1;char ch;int k=0;
  system("chcp 1251");
  cin.get(ch);
  while(ch!='\n')
  {
   if (ch=='('){ d1.push(ch);cin.get(ch);}
   else if (ch>='0' && ch<='9')
      {k=0; while (ch>='0' && ch<='9')   //генериране на число с повече цифри
           { k=k*10+ch-'0';cin.get(ch);}
            cout<<k<<" ";
       }
      else { if (ch=='+'|| ch=='-'|| ch=='*'|| ch=='/') d1.push(ch);
            else  if (ch==')') {char c=d1.top();
                              if (c=='+'|| c=='-'|| c=='*'|| c=='/') {cout<<c<<" ";d1.pop();c=d1.top();}
                  if (c=='(')  d1.pop();
                 }
            cin.get(ch);
          }
    }
}
```

2.2. Пресмятане на стойността на израз в посфиксен запис.

8 1 + 7 4 - - 11 9 - /

Алгоритъм:

1. Прочетете символ

2. Действия, които трябва да се извършат след всеки вход

Операнд (2.1) Включете го в стека и след това преминете към стъпка (1)

Операция (2.2) Вземете елемента от върха на стека, това е
втория операнд за операцията
(2.3) Извадете го от стека
(2.2) Вземете елемента от върха на стека, това е
първия операнд за операцията
(2.3) Извадете го от стека
(2.4) Приложете операцията върху операндите.
(2.1) Включете резултата в стека и след това преминете към стъпка (1)
(2.2) Изведете на екрана елемента от върха на стека
(2.5) СТОП

Нужен ни е стек от цели числа.

```
//изразът в посфиксен запис се въвежда с интервал след всеки
елемент и завършва с #
#include <iostream>
#include <stdlib.h>
```

```
#include <stack>
using namespace std;
int main()
{  stack <int> d1;
   char s;
   system("chcp 1251");
   cin.get(s);
   while(s!='#' )
  {
    if (s=='+'|| s=='-'|| s=='*'|| s=='/')
                              { int op2=d1.top();
                                d1.pop();
                                int op1=d1.top();
                                d1.pop();
                                int k=0;
                                switch (s)
                                {
                                  case '+':k=op1+op2;break;
                                  case '-':k=op1-op2;break;
                                 case '*':k=op1*op2;break;
                                 case '/':k=op1/op2;
                                }
                                d1.push(k);cin.get(s);
                               }
     else { int x=0;
            while (s!=' ')
            {
             x=x*10+ s-'0';
             cin.get(s);
             }
            d1.push(x);
           }
      cin.get(s);
   }
  cout<<d1.top()<<endl;
}
```

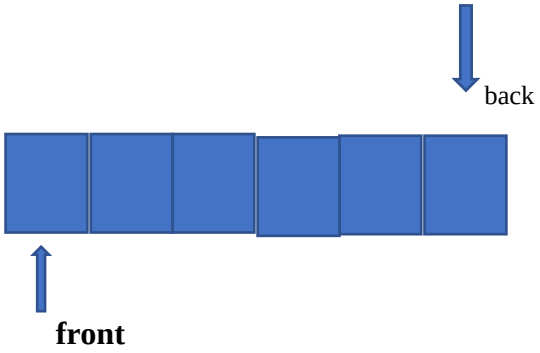
Опашка. Приоритетна опашка.

Опашката (на английски: *Queue*) в програмирането е вид абстрактна структура от данни (АСД) и е представител на абстрактните типове данни (АТД). Опашките спадат към линейните (списъчни) структури от данни, заедно със списъците и стековете. Опашката представлява крайно, линейно множество от елементи, при което елементи се добавят само най-отзад (enqueue) и се извличат само най-отпред (dequeue). Абстрактната структура опашка изпълнява условието „първият влязъл първи излиза“ (FIFO: First-In-First-Out). Това означава, че след като е добавен един елемент в края на опашката, той ще може да бъде извлечен (премахнат) единствено след като бъдат премахнати всички елементи преди него в реда, в който са добавени. *Опашка* е третата

от най-базовите абстрактни структури, които ще срещате отново и отново (другите две са Списък и Стек).

Структурата опашка и поведението на нейните елементи произхождат от ежедневната човешка дейност. Тя поддържа точно операциите, които са ни нужни да симулираме опашка в магазин - нареждане на хора в края на опашката и обработването (и премахването) им в началото. Така се поддържа нещо като "приоритет" - най-ранно влезлите елементи излизат най-рано. Този приоритет се задава имплицитно чрез реда на вкарване в опашката. Други примери за опашка са документи, чакащи да бъдат отпечатани или ескалатор превозващ хора. По този начин опашката изпълнява функцията на буфер.

По-късно ще разгледаме и структурата Приоритетна Опашка, при която можем да зададем приоритета експлицитно.



queue

```
#include <queue>
```

Поддържа стандартните операции за опашка.

- `.front()` - връща елемента в началото на опашката *без да го премахва*.
- `.back()` - връща елемента в края на опашката *без да го премахва*.
- `.push(T val)` - добавя нов елемент със стойност *val* в края на опашката.
- `.pop()` - премахва елемента, намиращ се в началото на опашката.
- `.size()` - връща колко елемента има в опашката.
- `.empty()` - връща `true` или `false` в зависимост дали опашката е празна или не.

Приложение

Структурата данни има най-различни приложения в компютрите - от пращане и получаване на съобщения до алгоритми в графи.

Тази структура данни се ползва в един от най-разпространените алгоритми с графи - Търсене в Ширина - като това я прави и една от най-често ползваните. Тя ни трябва в случаите, когато искаме да обработваме дадени събития или обекти в реда, в който са възникнали. Допълнително ползваме опашка като част от по-сложни алгоритми и структури данни.

Задача 1. Дадена е опашка q. Да се изключи от q най-малкият ѝ елемент, като всички останали елементи останат в опашката (не непременно в първоначалния ред).

```
#include<iostream>
#include<queue>
#include<stdlib.h>
using namespace std;
int main()
{
 queue <int> q;
 int n, s,i=0;
 system("chcp  1251");

 cout<<"Брой елементи:";
 cin>> n;
 do
 {
  i++;
  cin>>s;
  q.push(s);
  }
 while (i<n);
 int min=q.front();i=2;q.pop();
 while (i<=n)
  {
    if (min<= q.front()){q.push(q.front());q.pop();}
    else {q.push(min);min=q.front();q.pop();}
    i++;
   }
   queue <int> q1=q;
   for (int i=1;i<n;i++)
    {cout<<q1.front()<<"  ";
     q1.pop();
     }
 }
```

Задача 2. (Сортиране на опашки с пряка селекция) Да се подредят елементите на опашка в нарастващ ред.

Решение: Използваме нова опашка и прилагаме предната задача върху дадената опашка докато свърши, а минималните елементи поставяме в новата опашка.

Един специален вид опашка е цикличната опашка. При нея началото и края се свързват (след последния елемент отново следва първия). За разлика от стандартната опашка, в цикличния вариант първият елемент може да не е на нулева, ами на произволна позиция.

Задача 3. Да се напише програма, която въвежда имената на n деца и брой думи на броенка. Програмата извежда редът, в който децата ще излизат при броенката.

```
#include<iostream>
#include<queue>
```

```
#include<stdlib.h>
using namespace std;
int main()
{
 queue <string> q; string s;
 int n, br;
 system("chcp  1251");
 cout<<"брой деца: ";cin>>n;
 for (int i=1;i<=n;i++)
 {
   cin>>s;  q.push(s);
  }
 cout<<"брой думи: ";cin>>br;
 for (int i=1;i<=n;i++)
  {
   for (int j=1;j<br;j++)
    {
      q.push(q.front());q.pop();
     }
   cout<<q.front()<<endl;
   q.pop();
  }
}
```

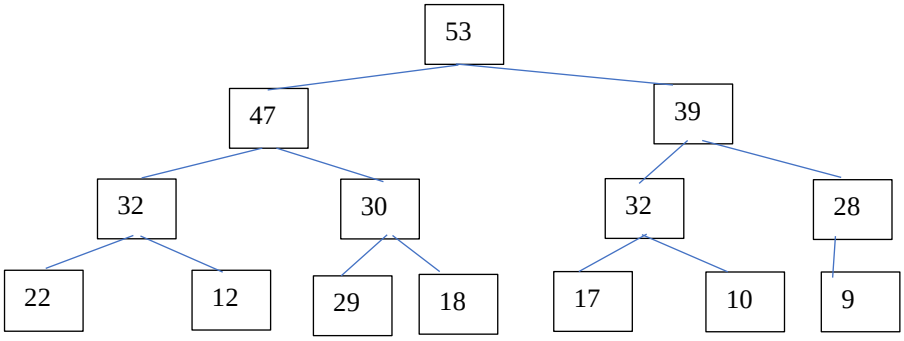
Приоритетна опашка (priority_queue)

```
#include <queue>
```

template< class T, class Container = std::vector<T>, class Compare = std::less<typename Container::value_type>
> class priority_queue;

Много полезна структура данни е приоритетна опашка. Тя е сравнително лесна както за разбиране, така и за писане, като в същото време е много полезна поради ефективността на операциите си и многобройните си приложения в задачи от реалния живот.

Може да се визуализира като пирамида, на която на върха е най-големия елемент (по някакъв критерий). Всички наследници са или равни или по-малки от родителя си.



Приоритетна опашка е реализирана в стандартната библиотека на C++. Тя се намира в хедъра <queue> и е от вида priority_queue<T>. В нея можете да добавяте елементи със сложност O(log(N)) и да взимате максималния елемент с O(1). Също така можете да премахвате максималния елемент със сложност O(log(N)). Често можете да я срещнете и като heap в английската литература.

Имаме голям брой конкуриращи се обекти, всеки от които има даден приоритет (важност), с който трябва да го обработим. В процеса на работа могат да се появяват нови обекти със свой собствен приоритет (потенциално много нисък или много висок). От нас се иска да можем бързо да:

- Добавяме нов обект с даден приоритет;
- Намираме обекта с най-голям приоритет;
- Премахваме обекта с най-голям приоритет.

Употреба

Поради ефективната си реализация и често възникващите проблеми от реалния живот, които изискват приоритизиране на дадени операции, тази структура данни намира редица приложения.

- `.top()` - връща максималният елемент от приоритетната опашка *без да го премахва*.
- `.push(T val)` - добавя нов елемент със стойност *val* в приоритетната опашка.
- `.pop()` - премахва максималния елемент от приоритетната опашка.
- `.size()` - връща колко елемента има в приоритетната опашка.
- `.empty()` - връща `true` или `false` в зависимост дали приоритетната опашка е празна или не.

За да демонстрираме употребата ѝ ще ползваме

Задача 4. Да се напише програма, която въвежда *N* числа и печата нечетните от тях в намаляващ ред.

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
 int N;
 priority_queue <int>q;
 cin>>N;
 for(int i=0;i<N;i++)
  {
   int val;
   cin>>val;
   q.push(val);
```

```
   }
 while(!q.empty())
 {
  if (q.top() %2) cout<<q.top()<<" ";
  q.pop();
 }
 return 0;
}
```

Задача 5. Да се напише програма, която въвежда информация за студенти - име, среден успех. Програмата подрежда студентите по среден успех. При еднакъв среден успех, студентите се подреждат по азбучен ред на името.

```
#include <iostream>
#include <stdlib.h>
#include <queue>
using  namespace std;
class Student {
      public:
        int facnom;
        string  name;
        double  ave;
        void readStudent (){
               cout<<"име:"; cin>>name;
               cout<<"среден успех:";cin>>ave;
               cout<<"фак. номер:"; cin>>facnom;
                           }
                 };
class  StudentComparer {
    public:
      bool  operator () ( Student x, Student y)
       {    return ((x.ave != y.ave) ? x.ave <  y.ave :
                                       x.name > y.name);
        }
                        };

int  main()
{
 priority_queue <Student, vector <Student>, StudentComparer>
q;
  int n;     Student s;
  system ("chcp 1251");
  cout<<"Брой студенти: "; cin>>n;
  for (int i=0;i<n; i++)
     {
       s.readStudent();
       q.push(s);
     }
  while (!q.empty())
   {
    cout<<q.top().facnom<<"  "<<q.top().name<<"  "
```

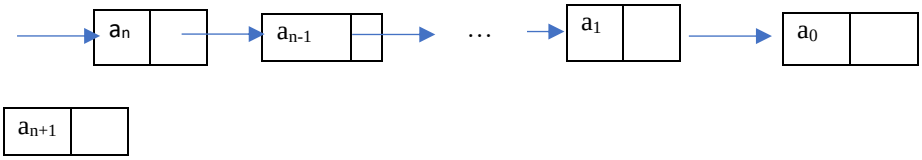
```
        <<q.top().ave <<endl;
    q.pop();
    }
}
```

Дек се нарича списък, при който всички включвания и изключвания на елемент може да бъдат извършвани от двата края на списъка. Идва от английското наименование Deque, което пък буквално означава опашка с два края (double-ended queue). Декът е на практика по–рядко използвана структура от данни в сравнение със стека и опашката. В стандартната библиотека на C++ (STL) стековете и опашките са реализирани всъщност чрез използване на дек.

Свързан списък

Най–просто можем да си представим списъка като наредена последователност (редица) от елементи. Нека вземем за пример списък за покупки от магазин. В списъка можем да четем всеки един от елементите (покупките), както и да добавяме нови покупки в него. Можем също така и да задраскваме (изтрием) покупки или да ги разместваме.

Свързаният списък е крайна редица от елементи от един и същ тип. Операциите включване и изключване са допустими в произволно място на редицата, т.е. списъкът е динамична структура от данни. Възможен е пряк достъп до елементите в началото и края на редицата и последователен до всеки от останалите елементи.



Свързан списък — това е структура от данни, при която последователните елементи се съхраняват на различни места в оперативната памет, а не в последователни полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент (указателят е неразделна част от самия елемент). Донякъде наподобява верига:

- всяко звено е свързано със следващото;
- за да има достъп до целия списък, нужен е единствено указател към първия елемент (другите се достигат от него);
- ако бъде сменен указателят на произволно звено, до всичко след него вече няма да има достъп (все едно да го откачим от веригата).

Основни операции

създаване на празен списък;

добавя елемент x на позиция pos;

изтрива елемент на позиция pos;

търси дали елемент x се среща в списъка;

Лесно можем да обходим всички елементи от структурата - почвайки от първия, следваме неговия указател към следващия, от там към по-следващия и т.н. до маркера за край. Обикновено стойността NULL в указателя за следващ елемент се приема за маркер за край. Допълнително се пазят два указателя - към първия и към последния елемент на списъка, което позволява лесно добавяне на елемент както в началото, така и в края на списъка.

Понякога искаме да може да се движим и назад в списъка. Ако всеки елемент пази указатели към предишния и следващия елемент, нарича се двусвързан списък, иначе - едносвързан. Двусвързаният списък позволява обхождане напред и назад.

Един от начините за работа със списъци е чрез използване на класа list от стандартната библиотека с шаблони Standard Template Library(STL).

list в STL

#include <list>

Деклариране на променлива:

list <*тип* > *име*; - създава празен списък

list <*тип* > *име(num, val)*; - създава списък с **num** на брой елемента със стойност **val**

Пример:

```
#include <list>
using namespace std;
....
int main(){
...
  list <string> l(5,"spisak");
  list <int> ls;
return 0;
}
```

Поддържа стандартните операции за списък.

- . push_front(val)- вмъква елемента **val** в началото на списъка.
- . push_back(val)- добавя елемента **val** в края на списъка.
- . pop_front() - изтрива елемента, който е в началото на списъка.
- . pop_back() - изтрива елемента, който е в края на списъка.
- . front() - намира първия елемент на списъка.
- . back() - намира последния елемент на списъка.

Пример:

```
list<int> ls;
for( int i = 0; i < 5; i++ )
{     ls.push_back(i);   }
 cout << "The first element is " << ls.front()<< " and the
last element is " << ls.back() << endl;
```

На екрана ще се изведе:

```
The first element is 0 and the last element is 4
```

- . clear() - изтрива всички елементи.
- . empty() - проверка дали списъка е празен.

Функцията empty() връща стойност true , ако в списъка няма елементи и false – в противен случай. Например в посоченият код се използва empty() за стопиране на цикъла, в който се визуализира съдържанието на елемент от списъка и след това се трие.

```
list<int> v;
  for( int i = 0; i < 5; i++ )
  {
    v.push_back(i);
   }
/* Обхождане на списък */
    while( !v.empty() )
     {
      cout << v.back() << endl;
      v.pop_back();
      }
```

- . size () - намира текущият брой елементи в списъка.
- .unique() – изтрива последователност от повтарящите се елементи. Функцията unique() изтрива всички последователности от повтарящи се елементи от списъка, като остава само един от тях.

Забележка: Ако искате да премахнете всички повтарящи се елементи от списъка, то трябва да го сортирате най-напред.

- .swap(lst1,lst2) -размяна на стойностите на 2 списъка.

Пример:

```
list<string> lst1;
lst1.push_front("I'm in l1!");
list<string> lst2;
lst2.push_front("And I'm in l2!");
swap(lst1,lst2); //или lst1.swap(lst2);
cout<<"The first element in lst1 is "<<lst1.front()<< endl;
cout<<"The first element in lst2 is "<<lst2.front()<< endl;
```

Горният код извежда:

```
The first element in l1 is And I'm in l2!
The first element in l2 is I'm in l1!
```

- . remove(val) - изтрива всички елементи от списъка, чиято стойност съвпада с val.

Задача 1. Напишете програма, която създава списък от букви и изтрива всички букви E.

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

```
using namespace std;
int main)
{
  list<char> charList;
  string azb= "ABCDEFGHEIJE";
  for(int i=0; i < 12; i++ ) //създаване на списък
   {
     charList.push_front(azb[i]);
    }
 // Изтриване на всяко срещане на 'E'
   charList.remove( 'E' );
   while(!charList.empty() )// обхождане на списък
    {
      cout << charList.back() << endl;
      charList.pop_back();
      }
}
```

Задача 2. Напишете програма, която въвежда редица от n числа и създава списък, който разделя положителните от неположителните числа.

```
#include <iostream>
#include <stdlib.h>
#include <list>
using  namespace std;
int main()
{
  list<int> List;int n, x;
  cin>>n;
  for( int i=0; i < n; i++ )
   {
     cin>>x;
     if (x>0)List.push_front(x);
     else List.push_back(x);
    }
   while( !List.empty() )
   {
      cout << List.back() << endl;
      List.pop_back();
      }
}
```

- .begin(), .end()- указатели към началото и края на списъка.
- .rbegin(), .rend()- указатели към началото и края на списъка за обратно обхождане.
- .insert(), .erase()- вмъква/ изтрива на позиция.
- .splice()- прехвърля елементи от един списък в друг.
- . sort() – сортира списък на място.
- .merge()- слива подредени списъци.
- .reverse()- обръща списък.

```
#include<iostream>
#include<list>
using namespace std;
int main()
{ list <int > l; int i,n,k;
  cin>>n;
  for (i=0;i<n;i++) //създаване на списък
   {
      cin>>k;
      l.push_front(k);
    }
 //първи и последен елемент
cout<<l.front()<<" "<<l.back()<<'\n';
 list<int> :: iterator j,t ;      //деклариране на итератор
  //обхождане на списък и извеждане на елементите му
 for (j=l.begin();j != l.end();j++)
       cout << ' ' << *j;cout<<endl;
 //търсене на елемент, удовлетворяващ условие
 for (j=l.begin(); j != l.end();j++)
    {
      if (*j<= 100) break;
      }
   t=j;//запомняне на итератора
   //вмъкване на елемент след избран елемент
   l.insert(++j , 999);
   l.insert(t  ,  111);  //вмъкване  на  елемент  пред  избран
елемент
   for (j=l.begin();j != l.end();j++)
       cout << ' ' << *j;
  }
```

За да се избере правилната структура на данните за заданието, трябва да има известна представа какво ще се прави с данните. Знаем ли, че никога няма да разполагаме с повече от 100 единици данни наведнъж, или трябва понякога да обработваме гигабайта данни? Как ще прочетем данните? Винаги в хронологичен ред? Винаги сортирани по име? Случаен достъп до номер на запис? Винаги ли ще добавяме / изтриваме данни в края или в началото? Или ще правим много вмъквания и изтривания в средата? Трябва да се постигне баланс между различните изисквания.

Ако трябва да се четат често данни по 3 различни начина, се избира структура от данни, която позволява да се правят и трите неща не твърде бавно. Често най-краткото и просто решение за програмиране за дадена задача използва линеен масив.

Ако се съхраняват данните като абстрактна структура от данни, това улеснява превключването към друга основна структура на данните и обективно измерване дали е по-бързо или по-бавно.

Предимства недостатъци

В по-голямата си част предимството на масива е недостатък на свързания списък и обратно.

Предимства на масива (спрямо свързан списък)

- Индекс - Индексираният достъп до всеки елемент в масив е бърз. В свързан списък трябва да се премине от началото, за да се достигне желания елемент. Това прави операции като сортиране, двоично търсене и т.н. невъзможни, или поне неприемливо бавни.
- По принцип достъпът до елемент в масив е по-бърз от достъпа до елемент в свързан списък.
- Тъй като всяка клетка от списъка съдържа указател към следващата клетка, то всеки елемент заема повече памет, отколкото би заемала клетката в масив.

Предимства на свързания списък (спрямо масив)

- Преоразмеряване - Размерът е динамичен и за това се ползва памет пропорционална на елементите в него. Свързаният списък може лесно да бъде преоразмерен чрез добавяне на елементи без да се засягат останалите елементи в списъка; масивът може да бъде увеличен само чрез разпределяне на нова част от паметта и копиране на всички елементи.

- Вмъкване - Добавянето на елемент в началото и в края е с константна сложност. Елемент може лесно да бъде вмъкнат в средата на свързан списък: създава се нова връзка с указател към връзката след нея и предишната връзка се прави, да сочи към новата връзка. При масивите добавянето в края също е константна операция, но в началото отнема цели O(N) стъпки.

Забележка: Как да се вмъкне елемент в средата на масив. Ако масивът не е пълен, всички елементи след мястото или индекса в масива, в който ще се вмъкват, се преместват напред с 1, след което се вмъква новия елемент. Ако масивът е вече пълен и ще трябва, в известен смисъл, да се „преоразмери масива“. Новият масив трябва да бъде с един елемент по-голям от оригиналния масив, за да се вмъкне новия елемент, след което всички елементи на оригиналния масив се копират в новия масив, като се вземе предвид индекса, за да се вмъкне елемента.

Кога бихме ползвали списък пред масив?

Сравнително рядко, но все пак има случаи, в които списъкът е по-удачен избор от динамичен масив. Един вариант е когато наистина ни трябва да пестим паметта възможно най-много и все пак да имаме бързо добавяне на елементи.