

## 4. УКАЗАТЕЛИ. МАСИВИ. СИМВОЛНИ НИЗОВЕ

### 4.1. УКАЗАТЕЛИ

#### 4.1.1. Адреси и указатели

Вече изяснихме, че за всяка дефинирана променлива или константа в ОП се отделя определено място, което има свой адрес.

В C++ съществува и специална операция - "извличане на адрес", която се задава с унарния оператор `&`. Ако `x` е променлива, то резултатът от операцията `&x` е адресът на `x` в паметта. Операторът `&` не може да се прилага върху константи и изрази, т.е. конструкции от вида `&3` и `&(x+1)` са недопустими.

В C++ съществува и възможност за дефиниране на специален тип променливи и константи, наречени указатели, на които могат да се присвояват само адреси. Ако адресът на една променлива е присвоен на даден указател (адресът на променливата е станал стойност на указателя) казваме, че указателят **сочи** към тази променлива, а стойността на променливата е **съдържание** на указателя.

Съществува и друга специална операция - "извличане на съдържанието на указател", която се задава с унарния оператор `*`. Например, ако указателят `rx` сочи към променливата `x`, то `*rx` ще бъде стойността на `x`.

#### 4.1.2. Дефиниране и използване на указатели

Общият вид на дефиницията на един указател е следният:

```
Име_на_тип *Име_на_указател [=Стойност];
```

`Име_на_тип` е името на типа на съдържанието на указателя, зададен в дефиницията. `Стойност` е инициализиращата стойност, ако дефиницията е съпроводена с инициализация, и може да бъде адрес на клетка в паметта или име на указател. Ето няколко примера:

```
int x; //Дефиниция на променлива x
float *p1; //Дефиниция на указател p1 към тип float
double *p2; //Дефиниция на указател p2 към тип double
int *p3=&x; /*Дефиниция указател p3 към тип int и
инициализация на p3 с адреса на x*/
```

Указателите `p1` и `p2` не са инициализирани, а неинициализираните указатели обикновено имат стойност `NULL` или `0`. Тя може да бъде присвоявана на всеки указател, независимо от неговия тип, когато той трябва да се превърне в неуказващ никаква променлива. `NULL` може да се разглежда както като числова константата `0`, така и като логическата стойност `false`. Това позволява да се организира проверката дали даден указател е инициализиран или неинициализиран, например така:

```
if ( !p ) //Проверка за неинициализиран указател
```

### 4.1.3. Операции с указатели

По време на изпълнение на програмите могат да се променят както стойностите, така и съдържанието на указателите. Например:

```
int x=20;           //Дефиниране на променлива от тип int
int *px;           //Дефиниране на указател px към тип int
px=&x;             //Инициализация на указателя
*px +=2;          //Увеличаване съдържанието на px с 2
cout<<*px<<'\n';  //Извеждане съдържанието на px
cout<<x;           //Извеждане на стойността на x
```

Чрез израза `px=&x` на указателя `px` се присвоява адресът на променливата `x` (указателят `px` се насочва към променливата `x`). От този момент стойността на променливата `x` е съдържание на указателя `px`, което се означава като `*px`. В четвъртия ред, съдържанието на `px` се увеличава с 2 и чрез оператора `cout << *px;` в петия ред се извежда на екрана. Резултатът ще бъде 22 (началната стойност 20 беше увеличена с 2). В последния ред на екрана се извежда стойността на променливата `x`. Резултатът също е 22, тъй като указателят `px` сочи към променливата `x` и увеличаването на съдържанието на `px` с 2 представлява увеличаване на стойността на променливата `x` с 2.

Указателите могат да участвуват като операнди в следните аритметични и логически операции `+`, `-`, `++`, `--`, `=`, `==`, `>`, `>=`, `<`, `<=`, `!=`. Изпълнението на аритметични операции върху указатели е свързано с някои особености. Ако към стойността на указателя `p` се добави единица (ако се изпълни операцията `p=p+1;` или операцията `p++;`), то адресът-стойност `p3` на `p` няма да се увеличи с 1, а ще се увеличи с броя на байтовете, заети от съдържанието на `p`, т.е. новата стойност на `p`, няма да е адреса на следващия байт, а ще е адреса на следващата променлива.

**Програма 4.1.** Указател. Стойност на указател. Съдържание на указател.

```
#include <iostream.h>
void main()
{
    int a=1;
    int *p1; p1=&a; cout<<"a="<<*p1<<endl;
    *p1+=10; //Съдържанието на указателя може да се променя
    cout<<"a="<<*p1<<endl;
    cout<<p1<<endl; //Ст-та на указателя може да се извежда
    int b=2;
    p1=&b; cout<<"b="<<*p1<<endl;
    //Стойността на указателя може да се променя
    p1++; cout<<"p1="<<p1<<endl;
    p1+=2; cout<<"p1="<<p1<<endl;
    float c=50;
```

```
float* p2=&c; cout<<"p2="<<p2<<" *p2="<<*p2<<endl;
p2+=2; cout<<p2<<endl;
}
```

#### 4.1.4. Указатели и константи

Езикът С++ позволява да се дефинират указатели-константи и указатели към константи. И в двата случая се използва ключовата дума `const`, но се поставя на различни места. Когато се дефинира указател-константа, ключовата дума `const` се поставя непосредствено пред името на указателя, а когато се дефинира указател към константа, тя се поставя непосредствено пред името на типа. Съдържанието на указател към константа не може да се променя, но стойността му може да се променя. Ето няколко примера:

```
int i; //i е променлива от тип int
int *pi; //pi е указател към променлива от тип int
const int ci=5; //ci е константа от тип int със стойност 5
//cp е указател-константа към променлива от тип int
int *const cp=&i;
//pci е указател към константа от тип int
const int *pci;
//cpc е указател-константа към константа от тип int
const int *const cpc=&ci;
```

Указателят `cp` е указател-константа, т.е. неговата стойност не може да се променя. Следователно операции от вида: `cp++`; `cp--`; `cp += 2`; `cp = pi`; са недопустими. Но съдържанието на `cp` не е константа и то може да се променя, т.е. операции като `*cp = 2`; са допустими.

Указателят `pci` е указател към константа и неговото съдържание не може да се променя, т.е. операции като `*pci = 2`; са недопустими. Но на указатели към константи могат да се присвояват други указатели към константи, например: `pci = cpc`;

Указателят `cpc` е указател-константа към константа. Следователно не може да се промени нито стойността му, нито съдържанието му, т.е. не може да се присвои нова стойност нито на `cpc`, нито на `*cpc`.

Върху указателите към константи е наложено още едно ограничение, а именно: *указатели към константна не могат да бъдат присвоявани на обикновени указатели*. Това означава, че присвояването `pi = pci`; е недопустимо. Ако такова присвояване беше позволено, то чрез изрази от вида `*pi = 4`; щяха да могат да се променят стойности на константи (след присвояването `pi` сочи към константа).

**Програма 4.2.** Указател променлива, указател константа. Указател към променлива, указател към константа.

```
#include <iostream.h>
void main()
{
    int a=1, b=2;
```

```

//Указател-променлива, който сочи променлива
int *p1; p1=&a; cout<<"a="<<*p1<<endl;
*p1=10;    cout<<"a="<<a<<endl;
p1=&b;     cout<<"b="<<*p1<<endl;

//Указател-променлива, който сочи константа
const int c=2; //Дефиниране на целочислена константа
const int *p2; p2=&c;
cout<<"c="<<*p2<<endl;
//*p2 не може да се променя, защото е константа
p2=&b;    cout<<"b="<<*p2<<endl;
//p2=&b е допустима операция, защото p2 не е указател-
//константа
//Указател-константа, който сочи променлива
int * const p3=&a;
// p3 се инициализира при създаването, защото е указател-
// константа и адресът в него не може да се променя
cout<<"a="<<*p3<<endl;
*p3=100; cout<<"a="<<*p3<<endl;
//*p3 може да се променя, защото е променлива

//Указател-константа, който сочи константа
const int * const p4=&c;
//На p4 не може да се присвои нов адрес,
//на *p4 не може да се присвои нова стойност
}

```

#### 4.1.5. Указатели към void

Типът, който се задава в дефиницията на един указател, е информация за компилатора относно начина на обработване (интерпретиране) на съдържанието на указателя. Например, ако указателят `p` е указател към `int`, то `*p` са два байта, интерпретирани като число от тип `int`, чиито начален адрес е стойността на `p`. Ако обаче `p` е указател към `float`, то неговото съдържание не са два, а четири байта, интерпретирани като число от тип `float`. Освен това, както беше изяснено, че адресната аритметика (аритметиката с указатели) също е съобразена с типовете на данните, към които сочат указателите.

Когато типът на данните, към които сочи един указател е без значение, т.е. важна е стойността на указателя (адресът), а не неговото съдържание, указателят се дефинира като указател към типа **void**, например:

```
void *p;
```

Указателите към `void` са предвидени с цел един и същи указател да може в различни моменти да сочи към данни от различен тип. Но при

опит да се използва съдържанието на указател към `void`, компилаторът ще регистрира грешка, тъй като не е известно как трябва да се интерпретира неговото съдържание. Съдържанието на указател към `void` може да се извлече само след привеждане на типа на указателя (`void *`) до типа на съдържанието. Такова привеждане може да се осъществи по два начина: чрез явно преобразуване на типа или чрез присвояване на `void` указателя на указател от съответния тип. Тези два начина за извличане на съдържанието на указатели към `void` се илюстрирани в следния фрагмент:

```
int n = 25; // Дефиниране на променлива n
int *p1;    // p1 е указател към int
void *vp;   // vp е указател към void
vp = &n;    // Инициализиране на vp
cout << *vp; // ГРЕШКА!!!
//Достъп до *vp чрез явно преобразуване на типа на vp
cout << *((int *)vp) << endl;
//Достъп до *vp чрез друг указател
p1 = vp;    //Присвояване на указател p1
cout<<*p1<<'\n'; //Съдържанието на p1 (ще бъде 25)
```

След присвояването `vp = &n`; указателят `vp` сочи към променливата `n`, която е от тип `int`. Но при опит да се използва съдържанието на указателя `vp`, т.е. ако се употреби конструкцията `*vp`, компилаторът ще регистрира грешка, тъй като `vp` е указател към `void` и не е известно как трябва да се интерпретира неговото съдържание. Съдържанието на указателя `vp` може да се извлече или след привеждане на типа му до `int *`, или чрез използване на втори указател от тип `int *` (в примера това е указателят `p1`), на който се присвоява указателят `vp`.

#### 4.1.6. Указатели към указатели

C++ позволява да се дефинира и указател към указател (двоен указател), т.е. такъв указател, който сочи друг указател, както е показано в следния фрагмент:

```
int x=20; //Дефиниране и инициализация на променлива
int *p1=&x; //Дефиниране и инициализация на указател
int **p2; //Дефиниране на указател към указател
p2=&p1; //Присвояване адреса на указателя на p1 на p2
cout<<*p1<<endl; //Извеждане съдържанието на p1
cout<<**p2<<endl;
//Извеждане съдържанието на съдържанието на p2
```

Дефиницията `int **p2`; означава, че `p2` е указател към указател, който е указател към тип `int`. След присвояването `p2=&p1` се получава следната ситуация: `*p1` е 20, колкото е стойността на `x`. Но `*p2` е `p1`, следователно `**p2` е също стойността на `x`, т.е. 20.

Може да се дефинира и троен указател, например

```
int ***p3;
```

Тук p3 е указател към друг указател, например p2. p2 също е указател към друг указател, например p1. А p1 е указател към променлива от тип int. Няма пречки да се дефинира четворен, петорен и т.н. указател, ако е нужен такъв.

#### 4.1.7. Динамични обекти. Оператори new и delete

C++ предлага средства за създаване и унищожаване на обекти (променливи, масиви, обекти на класовете и др. – тук терминът "обекти" се използва в широк смисъл, а не в тесния смисъл на обектно-ориентираното програмиране) по време на изпълнение на програмите. Такива обекти се наричат **динамични**. Паметта, която се заема при създаването на динамични обекти, се освобождава при тяхното унищожаване и може да бъде използвана отново.

Динамични обекти се създават чрез оператора new, който има следния синтаксис:

```
Указател=new Тип_на_динамичния_обект [инициализатор];
```

Тип\_на\_динамичния\_обект може да бъде някои от основните типове (int, float, char и др.) или име на клас, т.е. тип, конструиран от потребителя. Класовете са разгледани в главата "Класове и обекти". Указател е указател към тип, съвпадащ с типа на създавания динамичен обект. new е функция и при успешно изпълнение тя създава динамичен обект от посочения тип и връща като резултат (стойност на функцията) адреса на новосъздадения обект. Този адрес се присвоява на Указател. Ако в ОП няма достатъчно за създаването на динамичния елемент, операторът new връща като резултат стойност NULL.

```
float *p; //Дефиниран е указател към float
p=new float; /*Създадена е дин. пром. от тип float и
адресът ѝ е присвоен на p*/
*p=3.4;
//Инициализирана е новосъздадената динамична променлива
float *q=new float(4.5);
```

В последния пример с един оператор се дефинира указател, създава се динамична променлива и тя се инициализира със стойност 4.5.

Динамични обекти се унищожават чрез оператора delete, който има следния синтаксис:

```
delete Указател;
```

Указател е указателя към динамичния обект, който трябва бъде унищожен. В резултат от изпълнението на оператора delete паметта, заемана от динамичния обект, се освобождава.

Създаването на динамични променливи може да бъде съпроводено с инициализация. Инициализиращите стойности се задават в скоби, както е показано в следния пример:

Нека разгледаме една примерна програма, илюстрираща действието на операторите new и delete:

### Програма 4.3. Създаване и унищожаване на динамични обекти

```
#include <stdio.h>
#include <stdlib.h> //Заради exit
void main()
{
    int *p1;          //Дефиниране на указател към int
    p1=new int;      //Създаване на дин. променлива
    if (!p1){        //Проверка дали е създадена
        printf("Няма достатъчно памет\n");
        exit;        //Прекратяване изпълнението на програмата
    }
    *p1 = 2;         //Присвояване на стойност на дин. променлива
    delete p1;      //Унищожаване на дин. променлива *p1
    float *p2;
    //Създаване и инициализация на дин. променлива
    if (!(p2=new float(2.5))) {
        printf("Дин. променлива не е създадена.\n"); exit;
    }
    else
        printf("Съдържанието на указателя p2 е %f\n", *p2);
    delete p2;      //Унищожаване на динамичната променлива *p2
}
```

## 4.2. МАСИВИ

Масивът е структура от данни, състояща се от множество последователно наредени елементи от един и същи тип, достъпът до които се осъществява чрез индекси. Масивите могат да бъдат едномерни или многомерни (двумерни, тримерни и т.н.), в зависимост от броя индекси, чрез които се адресират елементите.

### 4.2.1. Дефиниране и използване на едномерни масиви

Общият вид на дефиницията на едномерен масив е следният:

Тип на елементите Име\_на\_масива [Брой на елементите];

Дефиницията на масива определя името на масива, типа на елементите му и неговия размер (брой на елементите). Броят на елементите трябва да се задава чрез константа или константен израз, защото памет за масивите се заделя по време на компилация.

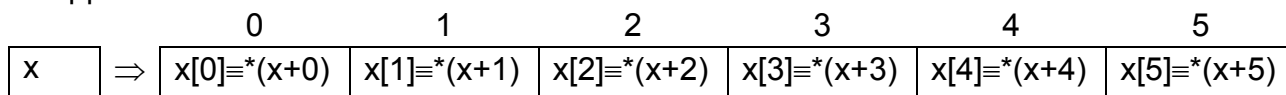
Дефиницията на едномерен масив с име x, състоящ се от 6 елемента от тип int, и на едномерен масив с име y, състоящ се от 5 елемента от тип float, има вида:

```
int x[6]; float y[5];
```

В резултат на дефиницията на даден масив компилаторът изпълнява следните действия:

- създава се указател-константа с името на масива и от типа на елементите на масива;
- заделя се място (поредица от байтове с последователни адреси) в ОП за елементите на масива;
- на създадения указател-константа се присвоява адресът на началния елемент на масива;
- елементите се номерират с последователни цели числа, наречени индекси. Номерацията (индексацията) започва от 0.

Например, дефиницията на масива  $x$  ще доведе до създаването на следното в ОП:



По дефиниция в  $a$  се намира адресът на 0-ия елемент. От правилата за операции с указатели следва, че  $x+i$  е адресът на  $i$ -ия елемент, а  $*(x+i)$  е съдържанието на адреса  $x+i$  или стойността на  $i$ -ия елемент, т.е.  $*(x+i)$  е един начин за указване на  $i$ -ия елемент. Съществува обаче и друг, по-удобен и по-често използван начин. Той е следният.

Отделните елементи се указват чрез името на масива и индекс, поставен в квадратни скоби, например  $x[0]$ ,  $x[1]$ ,  $x[2]$  са съответно нулевия, първия и втория елемент на масива  $x$ . Индексът може да бъде константа, променлива или израз от целочислен тип. Ако броят на елементите на масива е **6**, то индексите на елементите му трябва да бъдат от **0** до **5**. Компиляторът не контролира дали е удовлетворено това изискване, за това трябва да се грижи програмистът. Излизането на индексът от зададените граници обикновено води до грешки, които се откриват трудно.

Щом името на един масив има смисъл на указател, то вместо него може да се използва друг указател, който е инициализиран така, че да сочи първия елемент на масива. Например, ако  $p$  е указател към  $\text{int}$ , той може да бъде инициализиран с адреса на 0-ия елемент на масива  $x$  чрез всеки от операторите  $p=x$ ; и  $p=\&x[0]$ ; След тази инициализация, към 0-ия елемент на масива сочат два указателя - указателят  $x$  и указателят  $p$ . За достъп до елементите на масива може да се използва всеки от тях. Това означава, че записите  $*(x+i)$ ,  $x[i]$ ,  $*(p+i)$  и  $p[i]$  означават все  $i$ -тия елемент на масива  $x$ , т.е. те са еквивалентни.

Името на масив е указател-константа и поради това не са допустими операции, като  $x=r$ ,  $x++$  и  $x=\&x$ , които биха искали да променят стойността на този указател,.

Използването на масиви се илюстрира от следните програми:

#### **Програма 4.4.** Намира сумата на елементите на едномерен масив

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

```

int n, a[20];
printf("Задайте броя на елементите: ");
scanf("%d", &n);
//Въвеждане елементите на масива
printf("Въведете елементите на масива.\n");
for (int i=0;i<n;i++){
    printf("%d-и елемент: ", i);scanf("%d", &a[i]);
}

//Намиране сумата на елементите на масива
int s=0;
for (i=0; i<n; i++) s+=a[i];
printf("Сумата на елементите на масива а е %d\n",s);
}

```

**Програма 4.5. Намира индекса на най-големия елемент в едномерен масив**

```

#include <stdio.h>
void main()
{
    int n, a[20];
    printf("Задайте броя на елементите: ");
    scanf("%d", &n);
    //Въвеждане елементите на масива
    printf("Въведете елементите на масива.\n");
    for (int i=0;i<n;i++){
        printf("%d-и елемент: ", i);scanf("%d", &a[i]);
    }

    //Намиране индекса на най-големия елемент
    int ind_max=0;
    for (i=1; i<n; i++)
        if (a[i]>a[ind_max]) ind_max=i;
    printf("Индексът на max ел. в масива е %d\n", ind_max);
}

```

**Програма 4.6. Извежда индексите на елементите в едномерен масив с най-голяма стойност. Предполага се, че те са няколко**

```

#include <stdio.h>
void main()
{
    int n, a[20];
    printf("Задайте броя на елементите: ");
    scanf("%d", &n);
    //Въвеждане елементите на масива
    printf("Въведете елементите на масива.\n");
}

```

```

for (int i=0;i<n;i++){
    printf("%d-и елемент: ",i);scanf("%d",&a[i]);
}

//Намиране индекса на първия от елементите
int ind_max=0;
for (i=1; i<n; i++) if (a[i]>a[ind_max]) ind_max=i;

/*Извеждане индексите на елементите с най-голяма
стойност*/
printf("Най-голяма стойност %d ", a[ind_max]);
printf("имат елементите със следните индекси:\n");
for (i=ind_max; i<n; i++)
    if (a[i]==a[ind_max]) printf("%d ",i);
printf("\n");
}

```

**Програма 4.7.** Запомня и извежда индексите на елементите в едномерен масив с най-голяма стойност. Предполага се, че те са няколко.

```

#include <stdio.h>
void main()
{
    int n, a[20];
    printf("Задайте броя на елементите: ");
    scanf("%d",&n);
    printf("Въведете елементите на масива.\n");
    for (int i=0;i<n;i++){
        printf("%d-и елемент: ",i);scanf("%d",&a[i]);
    }
    int ind_max=0;
    for (i=1; i<n; i++) if (a[i]>a[ind_max]) ind_max=i;

    //Запомняне индексите на елементите с най-голяма ст-ст.
    int mas_ind[20],
        //Масив, в който се запомнят индексите
        j=0; //Индекс на елементите на масива mas_ind
    for (i=ind_max; i<n; i++)
        if (a[i]==a[ind_max]) mas_ind[j++]=i;
    /*Извеждане индексите на елементите с най-голяма
    стойност. */
    printf("Най-голяма стойност %d ", a[ind_max]);
    printf("имат елементите със следните индекси:\n");
    for (i=0; i<j; i++) printf("%d ",mas_ind[i]);
    printf("\n");
}

```

**Програма 4.8.** С два дадени масива а и b създава трети масив с елементи  $c_i = a_i + b_i$ .

```
#include <stdio.h>
void main()
{
    int n, a[20], b[20], c[20];
    printf("Задайте броя на елементите: ");
    scanf("%d", &n);
    //Въвеждане елементите на масива а
    printf("Въведете елементите на масива а.\n");
    for (int i=0;i<n;i++){
        printf("%d-и елемент: ", i);scanf("%d", &a[i]);
    }
    //Въвеждане елементите на масива b
    printf("Въведете елементите на масива b.\n");
    for (int i=0;i<n;i++){
        printf("%d-и елемент: ", i);scanf("%d", &b[i]);
    }
    //Намиране елементите на масива с
    for (i=0; i<n; i++) c[i]=a[i]+b[i];
    //Извеждане на елементите на масива с
    printf("Стойности на елементите на масива с:\n");
    for (i=0; i<n; i++) printf("C[%d]=%d\n", i, c[i]);
}
```

**Програма 4.9.** Извършва следните операции с елементите на даден масив:

- прехвърля положителните елементи в нов масив;
- намира средно-аритметичното на отрицателните елементи;
- запомня индексите на нулевите му елементи нов масив.

```
#include <stdio.h>
void main()
{
    int BrEl, BrPol, BrOtr=0, BrNul, mas_ind_nul[25];
    float a[25], mas_pol[25], SumaOtr=0, SredAr;
    printf("Задайте броя на елементите: ");
    scanf("%d", &BrEl);
    //Въвеждане елементите на масива
    printf("Въведете елементите на масива.\n");
    for (int i=0;i<BrEl;i++){
        printf("%d-и елемент: ", i);scanf("%f", &a[i]);
    }
    //Обработване на елементите на дадения масив
    int j=0,
    //Индекс на елементите в масива с положителни елементи
```

```

        k=0; /*Индекс на елементите в масива с индекси на
нулеви елементи*/
        for (i=0; i<BrEl; i++)
            if (a[i]>0) { mas_pol[j]=a[i]; j++; }
            else if (a[i]<0) { BrOtr++; SumaOtr+=a[i]; }
            else mas_ind_nul[k++]=i;
        BrPol=j; BrNul=k;
        SredAr=SumaOtr/BrOtr;
//Извеждане на резултатите
printf("Масивът с положителните елементи:\n");
for (i=0; i<BrPol;i++)
    printf("mas_pol[%d]=%f\n",i,mas_pol[i]);
printf("Масивът с индексите на нулевите елементи:\n");
for (i=0; i<BrNul;i++)
    printf("mas_ind_nul[%d]=%f\n",i,mas_ind_nul[i]);
printf("Средно-аритм. на отр. елементи: %f\n",SredAr);
}

```

#### 4.2.2. Дефиниране и използване на многомерни масиви

Общият вид на дефиницията на двумерен масив е следният:

Тип Име\_на\_масива [Брой на редовете][Брой на стълбовете];

Дефиницията на двумерен масив с име x, представляващ таблица от 5 реда и 10 стълба с елементи от тип int, има вида:

```
int x[5][10];
```

Отделните елементи се указват чрез името на масива и два индекса, поставени в квадратни скоби, например x[2][5] е елементът, който се намира на 2-ия ред и 5-ия стълб. Ако броят на редовете е **5**, то първият индекс може да бъде от **0** до **4**. Ако броят на стълбовете е **10**, то вторият индекс може да бъде от **0** до **9**.

Двумерният масив може да се разглежда и като масив от едномерни масиви и поради това той може да има елементи с един индекс. Например x[0], x[1], x[2] са нулевия, първия и втория ред на двумерния масив, т.е. всеки елемент е един едномерен масив.

Дефиницията на тримерен масив с име z, представляващ 3 таблици с по 5 реда и 10 стълба и с елементи от тип float има вида:

```
float z[3][5][10];
```

Този масив може да има елементи с един, два и три индекса.

- z[0], z[1], z[2] са нулевата, първата и втората таблици на тримерния масив z, т.е. всеки елемент на тримерния масив е един двумерен масив;
- z[0][0], z[0][1], z[0][2] са нулевия, първия и втория ред от нулевата таблица;
- z[0][1][0], z[0][1][1], z[0][1][2] са нулевия, първия и втория елементи от първия ред на нулевата таблица.

Елементите на масивите могат да участват във всички операции, разрешени за техния тип.

**Програма 4.10.** Програма за намиране сумата на елементите в отделните редове на двумерен масив.

```
#include <stdio.h>
void main()
{
    const m=3, //Брой на редовете
          n=2; //Брой на колоните
    int i, //Индекс, указващ реда в масива
        j, //Индекс, указващ стълба в масива
        A[m][n], //Зададения двумерен масив
        s[m]; //Масив с търсените елементи-суми
    //Въвеждане елементите на двумерния масив А по редове
    for (i=0;i<m;i++)
        for (j=0;j<n;j++){
            printf("A[%d][%d]=", i, j);
            scanf("%d", &A[i][j]);
        }
    //Сумиране елементите на двумерния масив А по редове
    for (i=0;i<m;i++)
        for (s[i]=0, j=0; j<n; j++) s[i]+=A[i][j];
    // Извеждане на масива-резултат
    for (i=0;i<m;i++)
        printf("s[%d]=%d\n", i, s[i]);
}
```

**Програма 4.11.** Програма за умножение на матрица по вектор.

```
#include <stdio.h>
void main()
{
    const N=3, //Брой редове на матрицата
          M=2; //Брой колони на матрицата
    int x[N][M], //Дефиниране на матрицата
        v[M], //Дефиниране на вектора-множител
        res[N]; //Дефиниране на вектора-резултат
    //Въвеждане стойностите на елементите на матрицата x
    printf("Въвеждане на матрицата x:\n");
    for ( int i=0; i<N; i++ )
        for ( int j=0; j<M; j++ ){
            printf("x[%d][%d]:", i, j);
            scanf("%d", &x[i][j]);
        }
    //Въвеждане стойностите на елементите на вектора v
    printf("Въвеждане на вектора v:\n");
```

```

    for ( i=0; i<M; i++ ){
        printf("v[%d]:",i); scanf("%d",&v[i]);
    }
// Умножение x * v и извеждане
for ( i=0; i<N; i++ ) {
    res[i]=0;
    for ( int j=0; j<M; j++ ) res[i]+=x[i][j]*v[j];
}
// Извеждане на елементите на вектора-резултат
printf("Елементите на вектора-резултат:\n");
for ( i=0; i<N; i++ )
    printf("res[%d]=%d\n",i,res[i]);
}

```

При работа с масиви трябва да се има предвид, че компилаторът не прави проверка дали стойностите на индексите, чрез които се адресират елементите, са в рамките на границите на масивите, зададени в техните дефиниции. Тази особеност крие опасност от допускане на трудно откриваеми грешки.

#### 4.2.3. Инициализиране на масиви

Един масив е инициализиран, ако при дефинирането му се зададат стойности на елементите му. Например:

```
int a[3]={1,2,3}; //Инициализация на масив
```

Това е дефиниция и инициализация на масив с име a, състоящ се от 3 елемента от тип int. Елементите му са инициализирани съответно със стойностите 1, 2 и 3, т.е. в момента на създаването на масива на елементите му се присвоени съответно стойностите 1, 2 и 3.

Допуска се броят на индексиралите (зададените начални) стойности да е по-малък от броя на елементите на масива, например

```
int a[5]={1,2,3}; //Инициализация на масив
```

В този случай масивът е частично инициализиран, получили са начални стойности само първите три елемента.

Когато дефиницията на един масив е съпроводена с инициализация, размерът на масива може да не се укаже , например:

```
double b[]={ 2.23, 4, -10.576, 12 };
```

В този случай размерът на масива b се определя от броя на инициализиращите стойности, т.е. размерът на масива b е 4, тъй като са зададени 4 инициализиращи стойности.

Двумерен масив се инициализира, като за всеки ред се задава съответно множество начални стойности, например:

```
int z[3][2]={
    {1, 1}, //Нулеви ред на масива, т.е. z[0]
    {2, 2}, //Първи ред на масива, т.е. z[1]
    {3, 3} //Втори ред на масива, т.е. z[2]
};
```

По аналогичен начин се инициализират и многомерни масиви с повече от два размера.

**Програма 4.12.** Намира броя на дните на зададен месец, като използва инициализиран масив.

```
#include <stdio.h>
void main()
{
    int M,G,D;
    //Инициализиран масив
    int DniPoMes[]={31,28,31,30,31,30,31,31,30,31,30,31};
    printf("Въведете номера на месеца:"); scanf("%d",&M);
    D=DniPoMes[M-1];
    if (M==2) {
        printf("Въведете номера на годината:");
        scanf("%d",&G);
        D+=G%400==0||G%4==0&&G%100;
    }
    printf("Броят на дните е %d\n",D);
}
```

Тук вместо инициализиран масив може да се използва масив-константа.

#### 4.2.4. Динамични масиви

Чрез оператора `new` могат да се създават и динамични масиви. Техните размери могат да бъдат и променливи и се задават в квадратни скоби след името на типа, указано в оператора `new`. Ето един пример:

```
int size; //Размер на масива
printf("Задайте размера на масива: "); scanf("%d",&size);
int *p= new int[size]; //Създаване на динамичен масив
. . . . .
. . . . .
delete [] p; //Унищожаване (изтриване) на динамичния масив
```

Обърнете внимание на това, че размерът на динамичния масив е променлива, чиято стойност се въвежда от клавиатурата. В общия случай размерите на динамичните масиви могат да бъдат изрази от цял тип. С други думи, размерите на динамичните масиви се определят по време на изпълнение на програмите и не е необходимо да бъдат известни по време на компилация.

Чрез оператора `delete[] p;` се унищожават динамичният масив. Обърнете внимание на скобите, поставени след оператора `delete`. Тези скоби означават, че указателят `p` сочи към масив (`p` съдържа началния адрес на масива), а не към един отделен обект. Независимо от това, дали операторът `delete` съдържа скоби или не, той ще освободи паметта за всички елементи на масива, щом указателят `p` сочи към масив.

Следователно, в горния пример унищожаването на динамичния масив може да стане и чрез оператора `delete p`;, който не съдържа скоби. Ефектът от скобите се проявява, когато елементите на масива са обекти от клас, който има деструктор (Класовете, обектите и деструкторите се разглеждат в главата "Класове и обекти"). В този случай, ако скобите липсват, деструкторът ще бъде изпълнен само за първия елемент.

**Програма 4.13.** Създава, въвежда и намира сумата на елементите на динамичен масив.

```
#include <stdio.h>
void main()
{
    int BrEl;
    printf("Въведете броя на елементите: ");
    scanf("%d",&BrEl);
    float* a;
    a=new float[BrEl]; //Създаване на динамичния масив
    if (!a) {printf("Масивът не е създаден.\n"); return;}
    //Въвеждане на елементите
    for (int i=0; i<BrEl; i++){
        printf("Въведете елемент %d: ",i); scanf("%f",&a[i]);
    }
    float s=0; //Намиране на сумата на елементите
    for ( i=0; i<BrEl; i++ ) s+=a[i];
    printf("Сумата на елементите е %f\n",s);
    delete []a;// Изтриване на динамичния масив
}
```

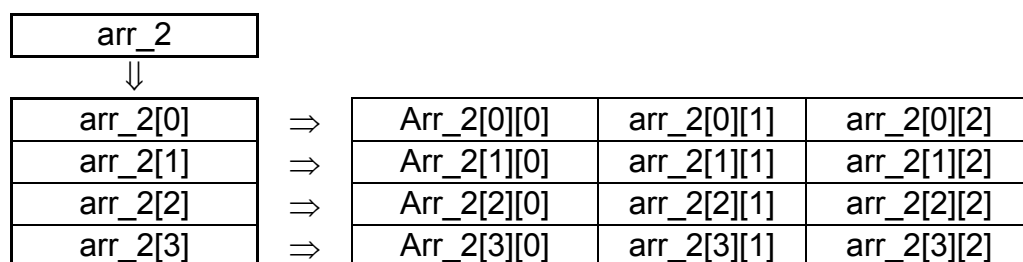
#### 4.2.5. Масиви от указатели. Конструирание на многомерни динамични масиви чрез масиви от указатели

Масивите от указатели са масиви, чиито елементи са указатели. Масив от указатели се дефинира по следния начин:

```
int *p[10]; //p е масив от 10 указателя към тип int
```

С помощта на масиви от указатели могат да се конструират многомерни динамични масиви.

Например двумерен динамичен масив `arr_2` с четири реда и 3 колони можем да конструираме както е показано на следната схема.



Фиг. 4.1. Динамичен двумерен масив, чрез масиви от указатели

### 1. Дефинираме и задаваме размерите на масива

```
int BrRed=4, //Брой на редовете на двум. масив
BrCol=3,     //Брой на стълбовете на двум. масив
i;          //Номер на ред
```

2. Дефинираме указател към началото на двумерния масив. Той трябва да е указател към указател (двоен указател), защото конструираме двумерен масив.

```
float **arr_2; //Начален адрес на двумерния масив
```

### 3. Създаваме масива от указатели

```
arr_2 = new float*[BrRed];
```

### 4. Създаваме масивите-редове – редовете на двумерния масив

```
for ( int i=0; i<BrRed; i++ )
    arr_2[i] = new float[BrCol];
```

Към отделните елементи на динамичния масив се обръщаме както към елементите на обикновен масив, т.е. `arr_2[0][1]` е елементът, който се намира на нулевия ред и в първата колона.

Изтриването на такъв масив трябва да стане в обратен ред на създаването му, т.е. най-напред трябва да се изтрият масивите-редове и след това масивът от указатели.

```
for (i=0; i< BrRed; i++)
    delete [] arr_2[i]; //Изтрив. на i-ия ред от двум. масив
delete [] arr_2; //Изтриване на масива от указатели
```

Следващата програма създава, използва и изтрива динамични масиви, конструирани чрез указатели.

**Програма 4.14.** Програма за умножение на матрица по вектор, използваща динамични масиви, конструирани чрез указатели.

```
#include <iostream.h>
#include <stdlib.h>
void main()
{
    int M,N, /*M-брой на редовете и N-брой на стълбовете
на матрицата*/
    m, n; //m-индекс на реда и n-индекс на стълба в матр.
    float **matr; /*Указател към 0-ия елемент на вектора
от улазатели към редовете*/
    float *vect; /*Указател към 0-ия елемент на вектора
vect*/
    float *p; /*Указател към 0-ия елемент на вектора-
произведение*/
    cout<<"Задайте размерите на матрицата:\n";
    cout<<"Брой на редовете: "; cin>>M;
    cout<<"Брой на стълбовете: ";cin>>N;
    cout<<endl;
    //Заделяне на памет за масива от указатели
```

```

    if ( !( matr=new float*[M] ) ) {cout<<"Няма достатъчно
памет\n";return; }
//Заделяне на памет за масивите-редове от матрицата
    for ( m=0; m<M; m++ )
        if ( !( matr[m]=new float[N] ) )
            {cout<<"Няма достатъчно памет\n";return;}
//Задаване на стойности на елементите на матрицата
    cout<<"Въведете елементите на матрицата:"<<endl;
    for (m=0; m<M; m++)
        for (n=0; n<N; n++) {
            cout<<"Елемент "<<m<<', '<<n<<":   ";
            cin>>matr[m][n];
        }
//Заделяне на памет за вектора vect
    if ( !( vect=new float[N] ) )
        {cout<<"Няма достатъчно памет\n";return;}
//Задаване на стойности на елементите на вектора
    cout<<"Въведете елементите на вектора:"<<endl;
    for (n=0; n<N; n++) {
        cout<<" Елемент "<<n<<": "; cin>>vect[n];
    }
//Заделяне на памет за вектора произведение
    if ( !( p=new float[M] ) )
        {cout<<"Няма достатъчно памет\n";return;}
//Изчисляване и извеждане елементите на произведението
    for (m=0; m<M; m++) {
        p[m]=0;
        for (n=0; n<N; n++)
            p[m]+=matr[m][n]*vect[n];
        cout<<p[m]<<"   ";
    }
/*Освобождаване на динамичната памет - става в обратен
ред на заделянето*/
//Освоб. на дин. памет, заета от редовете на матрицата
    for (m=0; m<M; m++)
        delete [] matr[m];
    delete [] matr; /*Освоб. на дин. памет, заета от
вектора от указатели. към редовете*/
    delete [] vect; /*Освобождаване на динамичната памет,
заета от вектора vect*/
    delete[] p; /*Освобождаване на дин. памет, заета от
вектора-произведение*/
}

```

**Програма 4.15.** За група окръжности и група точки определя по колко точки лежат във всяка окръжност.

```
#include <iostream.h>
#include <math.h>
void main()
{
    int i,j,BrO, // Брой на окръжностите
        BrT, // Брой на точките
        br[20]; // Брой точки във всяка окръжност
    double x[25],y[25],xc[20],yc[20],r[20];
    cout<<"Въведете броя на окръжностите: "; cin>>BrO;
    for (i=0; i<BrO; i++){
        cout<<"Въведете данните на "<<i<<"-та окръжност:\n";
        cout<<" xc= "; cin>>xc[i];
        cout<<" yc= "; cin>>yc[i];
        cout<<" r= "; cin>>r[i];
    }
    cout<<"Въведете броя на точките: "; cin>>BrT;
    for (j=0; j<BrT; j++){
        cout<<"Въведете данните на "<<j<<"-та точка:\n";
        cout<<" x= "; cin>>x[j];
        cout<<" y= "; cin>>y[j];
    }
    for (i=0; i<BrO; i++){
        br[i]=0;
        for (j=0; j<BrT; j++)
            if (pow(x[j]-xc[i],2)+pow(y[j]-yc[i],2)<=pow(r[i],2))
                br[i]++;
    }
    cout<<"Брой точки, съдържащи се в окръжностите:\n";
    for (i=0; i<BrO; i++)
        cout<<"В "<<i<<"-та окръжност има "<<br[i]<<
            " точки:\n";
}
```

### 4.3. МАСИВИ ОТ СИМВОЛИ И СИМВОЛНИ НИЗОВЕ

#### 4.3.1. Дефиниране на символен низ като масив от символи

В езиките C++ и C не е предвидена възможност за дефиниране на променливи от тип низ. Вместо променливи от тип символен низ тук се дефинират и използват масиви от символи, т.е. масиви, чиито елементи са от тип `char`, както е показано в следния пример

```
char x[10];
```

За да се съхрани един низ от  $n$  символа в масив от символи, дължината на този масив трябва да е не по-малка от  $n+1$  елемента, защото той трябва да побере  $n$ -те символа (колкото са в низа) и служебния символ `'\0'`, който служи за край на низа.

#### 4.3.2. Инициализация на масив от символи като низ

Един масив от символи може да се инициализира като низ по начина, по който се инициализира масив изобщо, но трябва да се инициализират всичките му елементи, в това число и допълнителният елемент, който се инициализира със стойност `'\0'`, т.е. както е показано в следния пример:

```
//Дефиниция и инициализация на масив s
char s[]={ 'a', 'b', 'c', 'd', '\0' };
```

За улеснение на програмистите, обаче, се предлага и следния, значително по-удобен начин:

```
char s[]="abcd";//Дефиниция и инициализация на масив s
```

Както се вижда, и при двата начина дължината на масива от символи се определя от дължината на инициализирания низ. Обърнете внимание, че при втория начин константата `'\0'` се поставя автоматично, докато при първия тя трябва явно да се посочи като стойност на последния елемент на масива.

Нека да си припомним, че инициализацията на масив е възможна само в момента на неговото създаване.

#### 4.3.3. Въвеждане и извеждане на символни низове

Символен низ може да бъде въведен и изведен по начина за въвеждане и извеждане на масив, т.е. чрез цикъл въвеждащ/извеждащ символите един по един като стойности на елементите на масива. Този начин обаче е неудобен.

Съществува по-удобна възможност за въвеждане на символен низ, чрез операторите:

```
scanf("%s",s); или cin >> s;
```

където  $s$  е масив от символи, но при следните две ограничения: размерът на въвеждания низ да не е по-голям от размера на  $s$ ;

- въвежданият низ не трябва да съдържа символите: интервал (' '), табулация ('\t'), нов ред ('\n') и връщане на каретката ('\r'), тъй като се третират като разделители.

Низ, съдържащ символа интервал, се въвежда чрез специална функция, към която можем да се обръщаме по следните начини:

```
a) gets(s);
```

Тази функция пренася символи от входния буфер (свързан клавиатурата) в символния низ  $s$ , докато срещне в буфера символа край на ред `'\n'`.

б) `cin.getline(s, n);` или `fgets(s, n, stdin);`

Тази функция пренася символи от входния буфер (свързан клавиатурата) в символния низ `s`, докато пренесе `n` символа или докато срещне в буфера символа край на ред `'\n'`.

в) `cin.getline(s, n, ',');`

Тази функция пренася символи от входния буфер в символния низ `s`, докато пренесе `n` символа или докато срещне в буфера символа запетая. Вместо запетая може да се зададе и друг символ. Когато се редуват въвеждане на символен низ чрез функцията `getline` и на числа чрез оператора `cin`, се налага синхронизация на вътрешния буфер с външното устройство. Тя се прави чрез оператора `cin.sync();`, който е обръщение към синхронизиращата функция `sync`.

Всеки символен низ може да се изведе чрез операторите

```
cout << s; puts(s); printf("%s", s);
```

#### **Програма 4.16.** Въвежда символни низове чрез `getline` и `fgets`.

```
#include <iostream.h>
#include <string.h>
void main()
{
    for (int i=0; i<3; i++){
        char Name[21], Adress[31]; int n;
        cout<<"Въведете име: "; cin.getline (Name,20);
        cout<<"Въведете адрес: "; cin.getline (Adress,30);
        cout<<"Въведете число: "; cin>>n;
        cin.sync();
        cout<<"Име:"<<Name<<endl<<"Адрес: "
            <<Adress<<endl<<"Число: "<<n<<endl;
    }
}
```

#### **Подобна програма, но с `fgets`**

```
#include <stdio.h>
#include <string.h>
void main()
{
    char Name[21], Adress[31]; int n;
    printf("Въведете име: "); fgets(Name,20,stdin);
    Printf("Въведете адрес: "); fgets(Adress,30,stdin);
    printf("Въведете число: "); scanf("%d",&n);

    printf("Име: %s\nАдрес: %s\nЧисло: %d\n",
        Name, Adress,n);
}
```

#### 4.3.4. Функции за работа със символни низове

C++ не предлага възможност да използваме оператора = за присвояване стойност на низ, както и операторите < <= > >= == != за сравняване на символни низове, но предлага цяла библиотека от функции във файла "string.h", чрез които можем да осъществим както тези, така и редица други операции. Тук ще покажем как можем да ползваме някои от тези функции.

##### 1. Копиране на символен низ в друг символен низ

Тази операция компенсира това, че при символните низове не може да се използва оператора за присвояване. На символен низ str1 може да се присвои стойността на символен низ str2 чрез функцията **strcpy(str1, str2)**; Низът str2 може да е променлива или константа.

**Програма 4.17.** Илюстрира използването на функцията strcpy.

```
#include <iostream.h>
#include <string.h>
void main()
{
    char string[10];
    char str1 []="abcdefghi";
    strcpy(string, str1);
    puts(string);
}
```

##### 2. Копиране на част от символен низ в друг символен низ

Чрез функцията **strncpy(str1, str2,n)**; може да се присвоят на str1 първите n символа от символен низ str2. Низът str2 може да е променлива или константа.

##### 3. Обединяване на символни низове

Тази операция се изпълнява от функцията **strcat(str1, str2)**. Тя разширява символния низ str1, като добавя към него символния низ str2.

**Програма 4.18.** Илюстрира използването на функция strcat.

```
#include <iostream.h>
#include <string.h>
void main()
{
    char obedinen[25]="\0";
    char blank []=" ", c []="C++",
    turbo []="Turbo"; //Създадени и инициализирани низове
    strcpy(obedinen, turbo);
    strcat(obedinen, blank);
    strcat(obedinen, c);
    puts(obedinen);
}
```

#### 4. Сравняване на символни низове.

Тази операция се изпълнява от функцията `strcmp(str1, str2)`; Тя връща цяло число. То е отрицателно, когато `str1<str2`, нула, когато `str1=str2` и положително, когато `str1>str2`.

#### 5. Намиране на дължината на символен низ.

Тази операция се изпълнява от функцията `strlen(str1)`. Тя връща цяло число, което показва броя на символите в низа. Тя не брои символът `'\0'`.

**Програма 4.19.** Илюстрира използването на функциите `strcmp` и `strlen`.

```
#include <iostream.h>
#include <string.h>
void main()
{
    char string1[10]="Иван", string2[10]="Иванка",
    string3[10]="Иван";
    cout<<strcmp(string1,string2)<<endl;
    cout<<strcmp(string2,string3)<<endl;
    cout<<strcmp(string1,string3)<<endl;
    cout<<"Дължината на низа string1 е "
    <<strlen(string1)<<endl;
}
```

#### 6. Превръщане на символен низ в число

В библиотеката `stdlib.h` се съдържат следните функции за превръщане на символен низ в число:

`atoi(s)` – преобразува низа `s` в число от тип `int`;

`atol(s)` – преобразува низ `s` в число от тип `long`;

`atof(s)` – преобразува низ `s` в число от тип `float`;

`_atold(s)` – преобразува низ `s` в число от тип `long double`.

Те са с един аргумент – символният низ, който трябва да се превърне в число и той трябва да съдържа само символи, допустими за съответния числов тип.

**Програма 4.20.** Илюстрира работата на функциите за превръщане на низ в число

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{
    cout<<atoi("1234")<<endl;
    cout<<atoi("1234.5")<<endl;
    cout<<atoi("12a4")<<endl; /*Превръща знаците до първия
нецифров знак*/
    cout<<atol("12345678")<<endl;
    cout<<atol("12345678.5")<<endl;
}
```

```

    cout<<atof("12.345")<<endl;
    cout<<_atold("12.3456789123")<<endl;
}

```

За същата цел може да се използва и функцията `int sscanf(char *s, char *format[, argum1]...);` където `s` е символен низ, а `format` съдържа форматни спецификатори, също както при функцията `scanf`. Разликата е, че вместо от стандартния вход `sscanf` чете символите от низа `s`. Например:

```

int i; float a; char bf[25], long int m;
sscanf("12 123.4", "%d %f", &i, &a); //i=12, a=123.4
sscanf("120 Hello", "%ld %s", &m, bf); // m=120 bf="Hello"

```

### 7. Превръщане на число в символен низ

В библиотеката `stdlib.h` се съдържат следните функции за превръщане на символен низ в число:

`itoa(i, s, b)` – превръща числото `i` от тип `int` в число с основа `b`, а след това в низ-стойност на `s`;

`ltoa(l, s, b)` – превръща числото `l` от тип `long` в число с основа `b`, а след това в низ-стойност на `s`;

`ultoa(ul, s, b)` – превръща числото `ul` от тип `unsigned long` в число с основа `b`, а след това в низ-стойност на `s`;

**Програма 4.21.** Илюстрира работата на функциите за превръщане на число в низ

```

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{
    char s[20];
    itoa(1234, s, 10); cout<<s<<endl;
    itoa(1234, s, 2);  cout<<s<<endl;
    ltoa(12345678, s, 10); cout<<s<<endl;
    ltoa(-12345678, s, 10); cout<<s<<endl;
    ultoa(1234567891, s, 10); cout<<s<<endl;
}

```

Превръщане на число в символен низ може да стане и с функцията: `int sprintf(char *s, char *format[, argum1]...);`

Тя е аналогична на `printf`, но извеждането става в низа `s`. Например:

```

int i=2; float a=4.5; char bf[25], long int m=4096;
sprintf(bf, "i=%d a=%f m=%ld", i, a, m);
//bf="i=2 a=4.5 m=4096"

```

### 4.3.5. Масиви от символни низове

Масив от символни низове може да се дефинира като двумерен масив от тип `char`. Например масив от 5 символни низа, всеки от които е с максимална дължина 11 символа (в това число и служебния символ за край на низа `'\0'`), се дефинира както следва

```
char s[5][11];
```

В този масив `char s[1]`, `char s[2]` и т.н. са отделни елементи.

Следващата програма дефинира такъв масив и въвежда и извежда елементите му.

**Програма 4.22.** Програма, която създава масив от символни низове, въвежда и извежда елементите на масива.

```
#include <iostream.h>
#include <string.h>
void main()
{
    const int n=5, //Размер на масива от символни низове
            m=21; //Максимална дължина на всеки символен низ
    // Дефиниране на двумерен масив от символи
    char arr_str[n][m];
    for (int i=0; i<n; i++) {
        cout<<"Въведете символен низ:";
        cin.getline(arr_str[i],m);
    }
    cout<<"Въведено:\n";
    for (i=0; i<n; i++)
        cout<<arr_str[i]<<endl; //Изв. i-ия симв. низ
}
```

Сега нека да си припомним, че могат да се създават масиви от указатели. Едно от приложенията на масивите от указатели са символните низове. Те се дефинират като масиви от указатели към `char`, като на всеки елемент на масива (указател към `char`) се присвоява низ. Това е възможно, тъй като вътрешното представяне на всеки низ (символи, оградени в двойни кавички) е указател-константа, който сочи към първия елемент на низа. След като масивът от указатели е инициализиран, с отделните низове се работи като с елементи на масива, т.е. достъпът до отделните низове се осъществява чрез индекс, съответстващ на позицията му в масива. Можем да направим така, че всеки указател да указва по един динамичен масив от символи (динамичен низ). Това може да се види в програма 4.22.

**Програма 4.23.** Програма, която създава масив от динамични символни низове, въвежда и извежда елементите на масива.

```
#include <iostream.h>
#include <string.h>
void main()
```

```

{
    const int n=2; //Размер на масива от символни низове
    int k; //Дължина на конкретен символен низ
    char bufer[21]; //Символен низ-буфер
    char *arr_str[5]; //Дефиниране на масив от указатели
    for (int i=0; i<n; i++) {
        cout<<"Въведете симв. низ:"; cin.getline(bufer,21);
        k=strlen(bufer); //Намиране броя на символи в i-и низ
//Създаване на памет за i-ия низ
        arr_str[i]=new char[k];
//Прехвърляне на i-ия низ в създаденото място
        strcpy(arr_str[i],bufer);
    }
    cout<<"Въведено: "<<endl;
    for (i=0;i<n;i++){
        cout<<arr_str[i]<<endl; //Извеждане на i-ия низ
        delete []arr_str[i]; //Изтриване на i-ия низ
        arr_str[i]=NULL; //Зануляване на указ. към i-и низ
    }
}

```

Горната програма извършва следните операции:

- дефинира (създава) масив от указатели;
- въвежда поредния низ в буфер, за да намери дължината му;
- създава място за поредния елемент на масива от низове във вид на динамичен низ с нужния размер;
- прехвърля въведения низ от буфера в създаденото място;
- извежда поредния елемент на масива от низове;
- освобождава мястото, заето от поредния елемент на масива от низове;
- занулява указателя към поредния низ.

Последната програма е по-ефективна от предпоследната, по две причини:

- разполага всеки елемент-символен низ в динамичен масив и го изтрива, когато престане да е необходим;
- всеки динамичен масив е толкова голям, колкото са символите на конкретния низ.