

## Лекция 6.

# Структурни абстракции на данните

*Могат ли програмите да  
обработват по-сложни данни?*

# Абстрактни структури от данни (АСД).

## Абстрактни типове данни (АТД)

Когато става дума за **съставни данни**, абстракцията на данните се използва като методология, която ни позволява да отделим (изолираме):

*начина на  
използване  
на съставния  
обект данни*

(1)

ОТ

(2)

*детайлите за  
начина на  
неговото констру-  
иране от по-прости  
обекти данни*

- (1) определя понятието **абстрактен тип данни** (АТД, *abstract data type*) – съвкупността от стойности на данните (областта) плюс спецификациите на операциите, използвани за създаване и обработка на данни
- (2) определя понятието **абстрактна структура от данни** (АСД, *abstract data structure*) – съвкупността от елементи на данните, чиято организация се определя от операциите за достъп, които се използват за съхранение и извличане на отделните елементи на данните, т.е. АСД имат две основни свойства:
  - могат да бъдат “разложени” на техните съставни части
  - подреждането на елементите е свойство на структурата, което се определя от това – какъв е достъпът до всеки елемент

# Съставни абстрактни типове данни: основни ПОНЯТИЯ

- Съставните (сложните, *composite*) типове са такива, чийто стойности се състоят от повече от една стойност
- Съставящите стойности се наричат **компоненти** (*components*) или **елементи** (*elements*)
- Компонентите принадлежат на други предварително определени типове. От тях по определени правила за структуриране (метод на структуриране) се конструират стойностите на сложните типове
- Доколкото типа на компонентите може също да бъде съставен, то може да се построи йерархия от структури, но крайните компоненти на всеки сложен тип са атомарни, т.е. са от прост тип и над тях могат да се извършват операциите, определени за съответния прост тип

# Съставни абстрактни типове данни: операции

За всеки АТД могат да са определени различни операции за обработка на стойностите му, но задължително за всеки метод за структуриране съответства **уникална двойка операции**, изобразяващи типа на компонентите в съставния тип и обратно:

- **конструктор** (*construction*) – **определя как съставната стойност се строи от стойностите на компонентите**
- **селектор** (достъп до елементите, *selection, access*) – **определя как стойностите на компонентите се извличат от съставната**




# Съставни абстрактни типове данни: базови операции

Върху съставните типове най-често се прилагат следните основни операции:

- *добавяне* на компонент (*add*)
- *премахване* на компонент (*remove*)
- *търсене, извличане и достъп* на компонент (*find, retrieve, access*)

Други операции разширяващи функционалността

- *проверка* дали има въобще компоненти (*is empty*)
- *премахване на всички компоненти* (*empty*) и др.

 Тези операции са важни по отношение на реализацията на произволен алгоритъм, поради което са имплементирани в различните езици като стандартни библиотеки пр. C++ STL

# Съставни абстрактни типове данни: класификация

Сложните типове най-често се класифицират според това дали:

- компонентите им са **подредени** – подредени и неподредени
- компонентите са стойности **от един и същи тип** – с еднотипни компоненти или и с разнотипни компоненти (комбинирани)
- **броят на компонентите** им е предварително фиксиран или не – статични и динамични
- според **механизмите за достъп** до компонентите им – с последователен (т.н. последователности) и с директен достъп

 Тези свойства са важни по отношение на тяхното използване при моделиране на данните

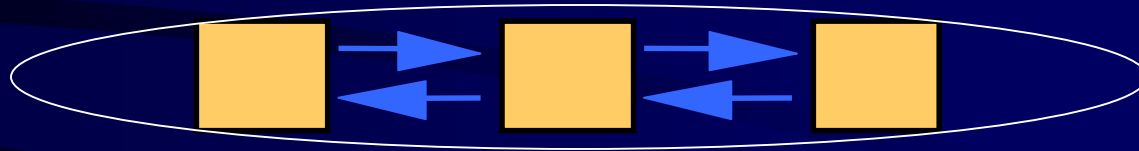
# АТД според подредеността на компонентите

АТД е **подреден** (структуриран) ако за всяка компонента е известно коя е предходната (или предходните) и коя е следващата (или следващите).

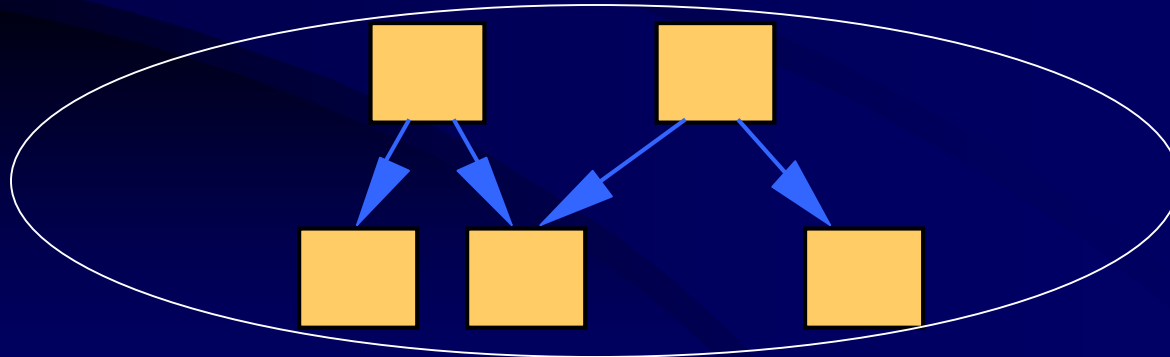
Ако реда на компонентите няма значение, т.е. елементите не са организирани по отношение един към друг, то съставният тип е **неподреден** (неструктуриран)

**Подредените АТД биват:**

- *линейни АТД* – всяка компонента има най-много един предходен и/или един следващ елемент



- *нелинейни АТД* – всяка компонента може да има произволен брой предходни и/или следващи елементи



# АТД според типа на компонентите

- **АТД с еднотипни компоненти:** всички елементи на съставната стойност са от един и същи тип

Типът на компонентите се нарича *базов тип* на съответния сложен тип

- **АТД с разнотипни компоненти:** компонентите на съставната стойност могат да принадлежат на различни предварително дефинирани типове



# АТД според начина на достъп до елементите



Класификацията “според начина на достъп до елементите” се отнася само за подредените АТД:

- АТД с **последователен** достъп: компонентите на съставната стойност могат да се обхождат единствено от началото към края (осигурено е средство, така че от всеки компонент може да се премине към следващ). Това означава, че за да извлечем някой елемент (да осъществим достъп до него) ще трябва да обходим всички елементи, намиращи се преди него

**Пример:** Представете си редица от стаи, като от всяка стая може да се преминава в следващата (преходни) и във всяка стая има един студент. За да намерим определен студент, който е в петата стая...

- АТД с **директен (пряк)** достъп: всеки компонент на съставната стойност е еднакво достъпен във всеки момент

**Пример:** Представете си множество от стаи и коридор, от който има врата към всяка стая

# АТД според броя на компонентите

- **Статични** АТД: броят на компонентите на съставната стойност е предварително фиксиран и не може да се променя по време на съществуване на структурата
- **Динамични** АТД: броят на компонентите на съставната стойност може да се променя по време на съществуване на структурата

**Забележка:** За динамичните АТД с последователен достъп е характерно, че елементи могат да се добавят или изключват от съставната стойност само в единия край на структурата, докато за тези с директен достъп това може да се прави на произволно място от структурата

 Много от съставните АТД са дефинирани в програмните езици като *вградени* съставни *типове*

# АТД масив: АМ

Всяка стойност от *тип масив (array)* представлява подредена крайна съвкупност от еднотипни компоненти, до които е осигурен пряк достъп. Масивът обикновено е статичен АТД

- **Характеристики:**
  - *краен* означава, че елементите са краен брой
  - *статичен* означава че размерът на масива трябва да се знае по време на компилирането, но това не означава че всички места в масива трябва да съдържат смислени данни
  - *подреден* означава, че относителното място на всеки елемент в масива е предварително дефиниран
  - *еднотипни компоненти* означава, че елементите са от един и същ тип данни
  - *прекия достъп* до елемент, означава, че при достъп до елемент на масива не е необходимо да се обръщаме към предишните елементи
- **Приложение:** Масивите се използват най-често за моделиране на подредени по някакъв признак еднотипни данни, които се подлагат на еднотипна обработка
- **Пример:** За студентите от даден курс често се налага да знаем кой студент е с най-висок среден успех, кои са студентите с отличен успех или какъв е средният успех на целия курс. В този случай е подходящо да се използва масив, като наредбата на компонентите (оценки на студентите) се извършва, например по поредния номер на студента в курса

# АТД масив: *Основни понятия*

- Типът на компонентите се нарича **базов тип** на масива
- Броят на признаците определящи наредбата на елементите се нарича *размерност на масива* (*dimension*). В зависимост от броят на признаците за наредба масивите се наричат съответно *едномерен*, *двумерен* и т.н., изобщо *многомерен масив*

**Пример:** В примера за студентите наредбата на компонентите се извършва по един признак. Възможно е наредбата да се определя от два признака (напр. оценките на студентите да са подредени по пореден номер студент и по изучавани дисциплини)

- Относителното място на един елемент на масива по отношение на някоя размерност се нарича **индекс на елемента** (*index*)

**Примери** за приложение на едномерни масиви: Те са естествена структура за съхранение на списъците от еднакви елементи на данните, като са списъци с покупки, ценови листи, списък с телефонни номера, списък с досиетата на студентите или списък от букви (низ)

# АТД масив: Схематично представяне

- Едномерен масив  
(вектор, таблица, *vector*, *table*)

пореден номер	4,25	6,00	...	...
---------------	------	------	-----	-----

- Двумерен масив

дисциплина \ пореден номер	Л А А Г	У П	А н г л и й с к и
	пореден номер		
1	3.00	6.00	6.00
2			
3			

- Тримерен масив

пореден №

дисциплина

г о д и н а

# АТД масив: Достъп и конструиране

- **Достъп:**

Мястото на елемента в масива е строго определено от набора на всичките му индекси по всички размерности на масива

Желаният елемент се посочва чрез неговия *индекс* (или индекси), което дава относителното му място в съвкупността

- **Конструиране:**

Конструирането на масивите, т.е. изграждането на съвкупността от стойностите на неговите елементи най-често се извършва поелементно, чрез *обхождане* (*traversing*) на масива. Това означава че ние последователно се обръщаме към елементите (чрез техните индекси) и записваме необходимите данни в тях

# Масивите в езиците за програмиране

В почти всички ЕП от високо ниво масивите са предварително дефинирани като стандартен тип, както и операциите за тяхното конструиране и достъп до всеки отделен елемент

- **Дефиниране (Python):**

```
myList=[1,2,3,4,5,6]
```

**Достъп** — винаги се използват индексите с различни нюанси:

- до отделен елемент :

```
myList[2]=100
```

- до сегмент:

```
myList[2:5]
```

- **Конструиране:**

- последователно поелементно конструиране (чрез присвояване):

```
myList=[]
```

```
for i in range(10):
```

```
myList[i]=1
```

- В съвременните езици от високо ниво Java и C# работата с масивите е аналогична, дори и синтаксиса се припокрива:

```
int[] anArray;
```

```
anArray = new int[10];
```

```
anArray[0] = 100;
```

# Масивите в езиците за програмиране

В Pascal, пример за декларация на два масива съответно от цели числа и булеви стойности и инициализация съответно със стойности 0 и false

**Var**

```
i      : Integer;  
myIntArray : Array[1..20] of Integer;  
myBoolArray : Array[1..20] of Boolean;
```

**Begin**

**For** i := 1 **to** 20 **do**

**Begin**

```
myIntArray[i] := 0;  
myBoolArray[i] := false;
```

**End;**

**End.**

 **Внимание!** В някои ЕП на индексите се съпоставят цели числа в интервал 0, ..., n-1 докато в други интервала е 1, ..., n



# АТД масив в C++ – ФМ и ЗМ

## Физически модел

Компонентите на един масив се записват в последователни клетки от паметта като първо се разгъва най-вътрешната размерност. Масивът се съхранява в толкова байта, колкото (изисква типа на компонента) \* (броя елементи)

## Знаков модел

В C++ масивът е статична структура, т.е. броят на компонентите му е фиксиран и се задава при декларирането на типа

- **Синтаксис** за деклариране на променлива-масив:

*<тип> <име> [ <константен израз<sub>1</sub>><sub>незад</sub> ] [ <константен израз<sub>2</sub>> ] ...*

, където:

- *<тип>* е име или декларация на тип и определя базовия тип на масива, т.е. типа на неговите компоненти. Той може да бъде всеки тип освен `void`, псевдоним или тип-функция
- *<име>* е името на променливата-масив
- броят на участващите елементи от вида [ *<константен израз>* ] определя размерността на масива
- всеки *<константен израз<sub>i</sub>>* е от целочислен тип и има стойност  $N_i \geq 0$  и определя броят на елементите на масива по съответната размерност (посл. с индекс  $= N_i - 1$ )
- *<константен израз<sub>1</sub>>* може да се пропусне само ако е извършено инициализиране на масива или ако той е деклариран като параметър, или в др. специални случаи

# АТД масив в C++ – примери

(1)

```
float x[10]; // дефинира масив, съдържащ 10 елемента
```

(2)

```
const int MAXCHILDREN = 500; // !препоръчва се  
double heights[MAXCHILDREN];
```

(3)

```
float matrix[10][15]; //двумерен масив matrix
```

(4)

```
struct {           // едномерен масив от структури  
    float x, y;  
} complex[100];
```

(5)

//декларацията може да е смесена – обикновени променливи и масиви

```
const int MAXPOINTS = 50;  
double xcoord[MAXPOINTS], xmax, ycoord[MAXPOINTS],  
       ymax, zcoord[MAXPOINTS], zmax;
```

(6)

Тип масив може да се декларира явно (typedef)

```
typedef      double OneDim [100];  
OneDim a,b;  
OneDim TwoDim [5]; //масив от масиви-двумерен масив
```

# АТД масив в C++ – примери


(7)

`int a[5][5][5];` // дефинира масив, съдържащ 5 реда, 5 колони и всяка колона има по 5 елемента

```
for( int i(0); i < 5; i++ )  
    for( int j(0); j < 5; j++ )  
        for( int k(0); k < 5; k++ )  
            a[i][j][k]; //достъп до  
елементите, чрез 3 индекса
```

# АТД масив в C++: Множество от стойности

- **Множество от стойности:**

- Всяка стойност от даден масив представлява съвкупност от стойности от базовия тип, на брой  $= N_1 * N_2 * \dots$  произведението на броевете по всички размерности на масива
-  На последователните елементи по всяка размерност  $I$ , чрез *<константен израз<sub>*i*</sub>>* се съпоставя индекс в интервала  $[0, N_i - 1]$

- **Примери:**

За променливите-масиви декларирани по-горе, множествата от стойности са както следва:

(1)

```
float x[10]; // масив, съдържащ 10 елемента от тип float
           //с индекси 0, 1, ..., 8, 9
```

(3)

```
float matrix[10][15]; //двумерен масив matrix със 150 елемента
                     //с индекси 0, 1, ..., 8, 9 и 0, 1, ..., 8, 14
```

(4)

```
struct {          // едномерен масив от структури със 100 елемента
    float x, y;
} complex[100];
```

# АТД масив в C++: Дефиниране на масив по време на изпълнение

Операторите `new` и `delete` съответно заделят (allocate memory) и освобождават (free memory) памет

```
#include <iostream>

int main() {
    using namespace std;
    int size, i = 0;
    cout<<"Number of elements";
    cin>>size;
    int* myarr = new int[size];

    for (i = 0 ; i < size ; i++)
        myarr[i] = 10;

    for (i = 0 ; i < size ; i++)
        printf_s("myarr[%d] = %d\n", i, myarr[i]);
    delete [] myarr;
}
```

# АТД масив в C++: Операция селектор

- **Селектор** – Директният достъп към компонентите на масив се осъществява чрез указване стойността на индекса или индексите им (при многомерните масиви), чрез постфиксната операция [...]:

*<име> [ <константен израз<sub>1</sub>> ] [ <константен израз<sub>2</sub>> ] ...*

, където:

- *<име>* е името на предварително декларирана променлива-масив
- броят на участващите елементи от вида [ *<константен израз>* ] е = размерността на масива според неговата декларация
- всеки *<константен израз<sub>i</sub>>* е от целочислен тип и определя индекса на елемента, до който се осъществява достъп по съответната размерност

- **Примери:**

```
x[4] = 0.5;  
cin >> x[4];  
value = x[n];  
cout << x[4];  
cin >> matrix[1][10];  
matrix[0][0] = 2.14;
```

# АТД масив в C++: Операции конструктор и инициализиране

- **Конструиране** – конструирането на масива може да се направи по два начина:

## 1. Чрез обхождане (*traversing*) на елементите:

```
for(int i=0; i<10; i++){  
    x[i] = float(i); // обхождане  
}  
double t_matrix[4][4];  
...  
for(int i = 0; i < 4; i++) {  
    for(int j = 0; j < 4; j++)  
        t_matrix[i][j] = 0.0; // по редове  
}
```

- **2. При инициализирането на масива**

```
float z[3] = {1.5, 6.8, 4.3};  
double mydata[4][5] = {  
    { 1.0, 0.54, 0.21, 0.11, 0.03 },  
    { 0.34, 1.04, 0.52, 0.16, 0.09 },  
    { 0.41, 0.02, 0.30, 0.49, 0.19 },  
    { 0.01, 0.68, 0.72, 0.66, 0.17 }  
};
```

# АТД масив в C++: инициализиране – особености

- При инициализирането на масива могат да се пропуснат вътрешните скоби {}:

```
double mydata[4][5] = {  
    1.0, 0.54, 0.21, 0.11, 0.03, 0.34, 1.04, 0.52, 0.16, 0.09,  
    0.41, 0.02, 0.30, 0.49, 0.19, 0.01, 0.68, 0.72, 0.66, 0.17  
};
```

- При инициализирането на масива ако не се зададе размер на масива компилаторът ще запълни масива според размера на инициализиращия списък:

```
int dayPerMonth[]={31,28,31, 30, 31, 30, 31, 31, 30,  
31,30,31}; // дефинира масив от 12 int елемента
```

- При инициализирането на многомерен масив може да не се зададе размерът само по първото измерение на масива:

```
double mydata[][5] = {  
    { 1.0, 0.54, 0.21, 0.11, 0.03 },  
    { 0.34, 1.04, 0.52, 0.16, 0.09 },  
    { 0.41, 0.02, 0.30, 0.49, 0.19 },  
    { 0.01, 0.68, 0.72, 0.66, 0.17 }  
};
```



# АТД масив в C++: Други операции

- Други операции с масиви **не са дефинирани**
- Т.е. всички други обработки с масив се извършват **поелементно** с използване на операцията селектор
- Тъй като елементите принадлежат на някой от базовите типове, то всички операции дефинирани за този базов тип са **приложими** върху елементите на масива

# АТД масив в C++: Особенности

- Както в C, така и в C++ масивите всъщност са последователно разположени клетки в паметта и поради това могат да бъдат обработвани и чрез използване на **указатели и аритметика** с тях

```
int main()
{
    int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30,
                    31, 30, 31 };

    int *pNumbers = number;

    cout << "\n*pNumbers: " << *pNumbers;
    cout << "\n*(pNumbers+1): " << *(pNumbers+1);
    cout << "\n*(pNumbers+2): " << *(pNumbers+2);
    cout << "\n*(pNumbers+3): " << *(pNumbers+3);
    cout << "\n*(pNumbers+4): " << *(pNumbers+4);
}
```

# АТД масив в C++: Особенности

- При работа с масиви има опасност да се излезе извън неговия диапазон. Трябва да се има пред вид, че **компиляторът не извършва проверки** и избягването на тези грешки е оставено на грижата на програмиста

```
double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28};  
cout << "2nd = " << distance[1] << endl;  
cout << "6th = " << distance[5] << endl; // компилатора не  
посочва грешка
```

- Могат да се правят т.нар. **“непълни” декларации на масиви** – без да се указва броят на елементите на масива по първата размерност. По-късно в рамките на същата област на видимост тази декларация се **попълва**:

```
int incomp[][3]; //непълна декларация на масив  
void main()  
{ ...  
    int incomp[4][3]; //тук вече е попълнена  
}
```

# Пример

**// деклариране и инициализиране на матриците**

```
const int N=3; //брой елементи
typedef double matrix[N][N]; //явно деклариране на тип масив
matrix a={1,0,0,0,1,0,0,0,1},b={1,0,0,0,1,0,0,0,1},product;
```

**// умножение на матриците**

```
for(int row = 0; row < N; row++)
    for(int col = 0; col < N; col ++) {
        double prdct = 0.0;
        for(int i = 0; i <N; i++)
            prdct += a[row][i] * b[i][col];
        product[row][col] = prdct;
    }
```

**// извеждане на резултата**

```
for( row = 0; row < N; row++) {
    cout<<endl;
    for(int col = 0; col < N; col ++)
        cout<<product[row][col]<<' ';
}
```

# АТД низ (последователност)

**АМ: Низът** (*string*) представлява подредена съвкупност от краен брой символи (знаци). Това е динамичен АТД, тъй като броят на неговите елементите (символите) може динамично да се променя по време на съществуване на структурата. До компонентите на низа е осигурен директен достъп

- **Основни понятия и характеристики:**

- Броят на елементите на даден низ се нарича **дължина на низа**
- **Базовият тип** на низа е символният тип
- По това, че компонентите на низа са еднотипни и по директния достъп до тях, този тип много прилича на едномерните масиви

- **Приложение:**

- Моделиране на данни, които представляват свързан текст на някой естествен език (основно приложение)
- Определяне на множеството на входни и изходни данни на описвания алгоритъм (въвеждане и извеждане на информация)

# АТД низ: Достъп, конструиране и други операции

- **Достъп:**  
Приликата между низовете и масивите определя и същия механизъм на избор – елементите на низа са достъпни пряко **чрез техните индекси** (относителното място в низа)
- **Конструиране:**  
Низовете се конструират просто **чрез изброяване** на символите (елементите на низа). Символите на един низ се ограждат в някакъв вид “*скоби*”, обикновено това са “...” или ‘...’
- За улеснение при използване на АТД низ се препоръчват следните **допълнителни операции**:
  - *преобразувател* (*transformer*) – операция, която чете стойности от файл или клавиатурата в низ
  - *наблюдател* (*observer*) – операция, която изпраща копие от низа към стандартното изходно устройство или файл

# АТД низ в C++: ФМ и ЗМ

- **Физически модел**

В паметта на компютъра всяка константа (стойност) от тип низ се записва в толкова байта, колкото са символите на низа +1

- **Знаков модел**

- По традиция в C++ / C няма собствен тип “текст-низ”. Обикновено се използва масив от тип `char` с **нула за край** – `'\0'` (*Null-terminated*)
- За да се опрости обработката на низове C++ предоставя набор от функции за обработка на низове, намиращи се във файла `string.h` и входно/изходни функции във файла `iostream.h` и `fstream.h`

- **Синтаксис за деклариране на променлива-низ:**

`char <име> [ <константен израз> незад ]`

, където:

- `<име>` е името на променливата-низ
- `<константен израз>` е от целочислен тип и има стойност  $N \geq 0$  и определя максималният брой знаци в низа, който поради допълнителния символ за край трябва да бъде с 1 по-голям от действителния брой знаци в низа
- `<константен израз>` може да се пропусне само ако е извършено инициализиране на низа или ако той е деклариран като параметър на функция

- **Примери:**

```
char s[7]; //низ с 6 елемента + '\0'
```

# АТД низ в C++: ЗМ – множество от стойности, операция селектор

- **Множество от стойности:**

Съвкупност от всички възможни последователности от символи на езика, дължината на които не надхвърля декларираната максимална дължина, включително и празната последователност

Последователности от символи, оградени от двойни кавички се приемат като **низови константи**: "qwerty", "%50 xyz", ""

- **Операцията селектор** за стойности от тип низ прилича на операцията за достъп до елементите **на** масив, осъществява се чрез индекса:

```
char strng[10];  
strng[0]='a';  
cout << strng[0];
```



# АТД низ в C++: ЗМ – конструктор и инициализиране

- **Конструктор и инициализиране** – по начин подобен на масивите:

## 1. Чрез обхождане (*traversing*)

```
char s[10];  
s[0]='a';   s[1]='b';   s[2]='\0';
```

## 2. При инициализиране – низовете могат да бъдат инициализирани по 2 начина:

(1) `char S_[4] = { 'a', 'b', 'c', 'd' };`

(2) `char S_null[5] = "abcd";` //или `char S_null[] = "abcd";`

🔊 При (1) в повечето случаи за правилната обработка на низа символът за край `'\0'` трябва явно да се зададе:

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

🔊 При (2) начин компилаторът сам добавя символа за край `'\0'` (препоръчва се)

# АТД низ в C++: Операции и релации

- **Въвеждане и извеждане:** Низовете могат да се въвеждат и извеждат с помощта на `cin` и `cout`, както и с помощта на `printf` и `scanf`
- **Всички останали операции и релации** с низове са реализирани като стандартни функции, чиито прототипи се намират в `string.h`. Най-важните от тях са:

Ф у н к ц и я	П р е д н а з н а ч е н и е
<code>strcat</code>	С лепва един низ към друг
<code>strchr</code>	Н а м и р а п ъ р в о т о с р е щ а н е н а д а д е н с и м в о л в н и з
<code>strcmp</code>	С равнява два низа
<code>strcpy</code>	К о п и р а е д и н н и з в д р у г
<code>strlen</code>	Н а м и р а д ъ л ж и н а т а н а н и з
<code>strncat</code>	Д о б а в я с и м в о л и о т е д и н н и з в д р у г
<code>strncmp</code>	С равнява символи от два низа
<code>strncpy</code>	К о п и р а с и м в о л и o t e d i n n i z v d r u g
<code>strrchr</code>	Н а м и р а п о с л е д н о т о с р е щ а н е н а д а д е н с и м в о л в н и з
<code>strstr</code>	Н а м и р а п ъ р в о т о с р е щ а н е н а д а д е н н и з в д р у г

- **Пример:**

```
strcpy(strng, "ab");
```



Запомнете, че присвояване на низове не е реализирано



Често срещана грешка е препълване на низа

# АТД низ в C++: Стандартни функции за преобразуване

- Стандартните функции, чрез които можем да извършваме преобразуване между тип низ и други стандартни типове са реализирани в `stdlib.h`. Най-важните от тях са:

Ф у н к ц и я	П р е д н а з н а ч е н и е
<code>atof</code>	П р е о б р а з у в а <code>string</code> в <code>float</code>
<code>atoi</code>	П р е о б р а з у в а <code>string</code> в <code>int</code>
<code>atol</code>	П р е о б р а з у в а <code>string</code> в <code>long</code>
<code>_itoa</code>	П р е о б р а з у в а <code>int</code> в <code>string</code>
<code>_ltoa</code>	П р е о б р а з у в а <code>long</code> в <code>string</code>
<code>_ecvt</code>	П р е о б р а з у в а <code>double</code> в <code>string</code>
<code>strtod</code>	П р е о б р а з у в а <code>string</code> в <code>double</code>

# АТД низ в C++: Стандартни библиотеки на C++

- STL съдържа класа `string`, който предоставя много по-сигурна реализация на абстракцията текст-низ на високо ниво, които са включени в заглавния файл `<string>` (и още няколко вида низове)
- MFC включва класа `CString`, който поддържа обработка на низове

# АТД низ в C++: Основни алгоритми с низове

- Замяна на подниз от даден низ с друг
- Изтриване на подниз от даден низ
- Търсене на подниз в даден низ
- Подреждане на низове в азбучен ред
- Определяне дали даден низ е палиндром.

**Палиндром** – “алена фанела”, "madam I 'm adam", "Able was I ere I saw Elba" (виж функцията `_strrev`)