
ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

# ИНФОРМАТИКА

## част втора

## Основи на програмирането на C++

**лектор: гл. ас. д-р Стефан М. Панов**

**Катедра “Информатика”**

# Лекция 1 (9)

# Указатели в C++.

Една от най-важните особености на езика C++ се явява използването на указатели. Благодарение на тях може да се изграждат свързани списъци, динамично да се заделя памет, както и да се даде възможност на функциите да променят своите аргументи.

## Определение на указател и инициализиране

Всеки обект на езика (променлива, константа, елемент на масив и т.н.) се съхранява в определени клетки от паметта на компютъра или казано другояче, на определен адрес.

**Променлива, чиято стойност е адрес от паметта, се нарича указател.**

За такива променливи казваме, че са от **тип указател**. Най-често адресът, който се съдържа в указателя обозначава местоположението в паметта на друга променлива. Т.е. стойността на указателя посочва (указва, оттам и името) местоположението на другата променлива и позволява косвен

достъп до стойността ѝ за разлика от прекият достъп с помощта на нейното име.

Както всяка променлива в езика C++, и указателят трябва да се дефинира преди да бъде използван. Става така:

**тип *име_на_указател** (може и **тип* име_на_указател**)

Елементът **тип** означава типът на обекта, към който указателят сочи, при това той **трябва да бъде допустим C++ тип**. Символът * тук има съвсем различен смисъл от тоя на оператора за умножение!

**Примери:**

```
int *pTemp;       //указател към променлива от тип int
double z, *p2;    // променлива z и указател към променлива от тип double
```

**След като се определи даден указател той трябва да бъде използуван едва след като е инициализиран, т.е. след като му е присвоена правилна стойност - валиден адрес от паметта.**

**Забележка 1**: Глобалните указатели и локалните статични указатели се инициализират **автоматично** с **NULL**. Какво е статична памет ще се разгледа в друга лекция. Желателно е локалните указатели да се инициализират с **NULL**. Нулевата стойност, използувана при инициализиране на указателите, има име **NULL**, дефинирано в стандартния заглавен файл **cstdio**. Указатели, които имат стойност **NULL** се наричат нулеви указатели и се счита, че не сочат към нищо. Всяка локална променлива, вкл. и указател, съдържа произволна стойност, ако не е инициализирана.

**Забележка 2**: Нищо не може да попречи на накараме (умишлено или несъзнателно) указателят да сочи **„някъде_си_в_паметта“**. Ето защо указателите са потенциално опасни. За да бъде безопасно използването на указатели трябва във всеки един момент да знаем към какво сочат те!

## Оператори, използвани с указатели

Има два специални унарни оператора за указатели, които ни позволяват да ги използуваме ефективно - “**&**” и “*****”.
Операторът & връща адреса на операнда посочена след него.

Използва се така: **&операнд** (операнд е **променлива** или **елемент на масив**)
Например, при изпълнение на следващата инструкция: **pTemp = &temp;**

в указателя **pTemp** се записва адресът на променливата **temp**. Когато се извлича адресът на многобайтова променлива, всъщност се извлича адресът на най-младшия байт на паметта, заемана от променливата. Ако приемем, че целочислената променлива **temp** заема памет с адреси от 5024 до 5027 то

| | **temp** | | | | | |
|---|---|---|---|---|---|---|
| | | | **47** | | | |
| 5024 | 5025 | 5026 | 5027 | 5028 | 5029 | 5030 |

**Табл. 9.1 – променливата temp с начален адрес 5024 има стойност 47**

в указателя **pTemp** ще се запише 5024. Неправилно е да се използува **&** за определяне на адрес на променлива от клас register.

**Другият специален оператор е *.** Използва се така: ***операнд** (операнд е указател) **и означава стойността на променливата, чийто адрес се съдържа в указателя.** Казано по друг начин * означава стойността на променливата, която се сочи от указателя.

Например, ако **val** е също променлива от тип **int**, то инструкцията

**val = *pTemp;** **//** изпълнена след **pTemp = &temp;**

ще ѝ присвои стойността на **temp**, защото нейният адрес се съдържа **pTemp**. Т.е. на **val** ще се присвои 47 за примера от Таблица 1. Инструкциите разгледани дотук в текущия раздел са в основата на следващата програма.

**Важно: Съгласно правилата на адресната аритметика, разгледани по-нататък, като операнд след * може да се използува и израз, съставен от указател и валидно аритметично действие.**

```
#include <iostream> // nashata 41-va programa
#include <cstring> // primer za izpolzvane na operatorite & i * s ukazateli
using namespace std;
int main() {
	int temp; // temperatura
	int val; // druga promenliva
	int *pTemp; // ukaztel
	temp = 47;
	pTemp = &temp;// inicializirame ukzatelia
	val = *pTemp;
	cout << "Temperaturata e " << val << " gradusa \n";
	return 0;
}
```

**Заключение:** конструкцията *указател е равностойна на директното използване на името на променливата, сочена от указателя. За нашия

пример навсякъде след инициализирането на pTemp можем да използваме *pTemp вместо temp.

**Още веднъж за символа звездичка “*”:**

- Използвана при дефинирането на променлива звездичката единствено указва, че променливата непосредствено вдясно от нея е указател.
- Използвана с указател след неговото дефиниране, звездичката е специален унарен оператор за указатели. Да се има предвид, че “**&**” и “*” имат по-висок приоритет от всичките пет **двуоперандни** аритметични оператори. Унарният минус има равен приоритет с тях.
- Използвана извън контекста на указателите звездичката е оператор за умножение.

## За важността на базовия тип на указателя

Когато се обявява указател, да речем указател към тип **int** компилаторът "предполага", че стойностите, към които сочи тоя указател имат тип **int**. Например, при изпълнение на следния код се издава съобщение за грешка:

**int *pi; double d; ... pi = &d; // GRESHKA!**

Изразът **&d** генерира указател към **double**-стойност, а **pi** е указател към тип **int**. Тези два типа са несъвместими, ето защо компилаторът сигнализира за грешка. Следващата програма показва как може да заобиколим проблема - чрез явно преобразуване на тип - но като правило от такова хитруване няма особена полза. Да, програмата работи, но грешно.

В инструкцията e = *pi; променливата e ще получи при присвояване само 4 байта (а не 8) заради типът на pi. Но самата стойност в променливата **d** заема 8 байта, именно затова резултатът при отпечатване с cout е грешен!

```
#include <iostream> // nashata 42-a programa
using namespace std;// ! primer za greshno raboteshta programa
int main() {
	double d=789.54,
	double e;
	int *pi; // ukaztel kym int
	pi = (int*)&d; // zaobikaliame chrez iavno prebrazuvane na tip
//	pi = &d; // dava syobshtenie za greshka
	e = *pi;// chrez *pi se prochitat samo 4 baita vmesto 8
	cout << "Stojnostta  e " << e << "\n";// tova e druga stojnost !
	return 0;
}
```

## Аритметични операции с указатели

Същността на указателите като **особен** вид променливи определя и кои от аритметичните действия са позволени при тях. **При адресната аритметика са валидни само операциите събиране и изваждане. При това, има изисквания и към операндите на тия операции**.

**Едноместните операции ++ и -- са напълно валидни за указателите**. Най-съществено е, че те притежават необходимата "интелигентност" и **увеличението (намалението) с 1 в действителност е увеличение (намаление) с толкова байта, колкото е типът на променливата-указател**.

Например, ако **pi** от предишната задача сочи към адрес 5000 (т.е текущата стойност на **pi** е 5000) след изпълнението на инструкцията **pi++;** новата стойност на **pi** ще бъде 5004, защото променлива от тип **int** заема 4 байта. Аналогично след **pi--;** новата новата стойност в **pi** ще бъде 4996.

Ако отново се върнем към масива от 20 целочислени елемента от лекция 7 и е изпълнена инструкция **pi = &temp[0]**; става ясно какво се случва след

| **temp[0]** | **temp[1]** | **temp[2]** | .... | **temp[19]** |
|---|---|---|---|---|
| 1024 | 1028 | 1032 | | 1100 |

**Табл. 9.2 – целочислен масив от 20 елемента**

инструкцията **pi++;** - **Указателят pi сочи към следващия елемент на масива. Т.е. към следващата целочислена стойност.**

Освен инкремент и декремент обикновените операции **събиране** и **изваждане** са също приложими към указатели, но вторият операнд може да има **само целочислени** стойности. Например **pi = &temp[0]**; **pi = pi +19;** ще насочи указателя към последния елемент на масива от табл. 9.2, т.е.

действат същите правила както при увеличение или намаление с 1. Казано обобщено:

**Всички аритметични операции над указатели се изпълняват относно базовия тип на указателя**.

Указател може да се използва и от лявата страна на инструкция за присвояване. Например следният код: **pi = &temp**; **pi = pi +2; *pi = 173;** ще насочи указателя към елемента **temp[2]** и ще му присвои 173. **Операциите ++ и -- са едноместни операции и могат да се комбинират с други операции.** При изпълнение на следващата инструкция **(*pi)++;** стойността на елемента **temp[2]** ще стане 174. Тука скобите са задължителни защото операторът инкремент има по-висок приоритет от оператора * за указатели!

Да разгледаме разликите между префиксната и постфиксна форма на ++ и –

***p++** е същото като ***(p++)** Но увеличението на **p** става след като се използва.

Т.е. първо се извлича стойността, сочена от **p**, и след това се увеличава указателят **p** да сочи към следващия адрес. (Аналогично за ***p--**.)

***--p** е същото като ***(--p)**

Първо се се намалява указателят **p** да сочи към предишния адрес и после се извлича стойността, сочена от **p**. (Аналогично за ***++p**.)

Още правила: Присвояването на указател на указател, ако те сочат към еднотипни обекти, е допустимо.

**Аритметичните операции - събиране на указатели едни с други, събиране или изваждане на указатели с числа от тип float или double, умножение или деление на указатели, са невалидни аритметични адресни операции.**

В случаите, когато два или повече указателя сочат към променливи, между които съществува връзка (например към различните елементи на масив), такива указатели могат да бъдат изваждани един от друг. Например, ако **p1** и **p2** са указатели към елементи от един и същи масив, разликата им **p1 - p2 - 1** ще определи броя на елементите между двата елемента, към които сочат **p1** и **p2**.

**Указатели могат да се сравняват** с използването на операторите за отношение **==**, **<** и **>**. Както и при изваждането това има смисъл само когато двата указателя сочат към променливи, между които съществува връзка!

## Указатели и масиви

В C++ указатели и масиви са тясно свързани. Да разгледаме следните 3 инструкции **char str[80]; char *ps; ps = str;** от които интересната е третата.

Използването на името на масива е равносилно на обръщение към адреса на първия елемент на този масив. Затова еквивалентна на третата е следната инструкция: **ps = &str[0];** Оттук нататък в програмата и указателят **ps** и името на масива **str** могат да бъдат използвани за достъп към елементите на масива. Например достъпът към шестия елемент става с **str[5]** или ***(ps+5)**. Малките скоби отново са заради по-високия приоритет на * в сравнение със събирането. Нещо повече, **указател, сочещ към масив, може да се индексира все едно той е името на масива, тоест ps[5] също е валидно** (което още веднъж говори за тясната връзка между указатели и масиви). Но и името на масива може да се използва с оператора * затова и ***(str+5)** е вярна конструкция. Това е показано и в програма_43.

При употреба на средни скоби говорим за индексиране на масив или указател, а във втория случай за аритметични операции с указатели. Вторият подход е по-бърз затова се предпочита в програмите където

бързодействието е от значение. Но защо вторият вариант е по-бърз: В процеса на компилация всички конструкции от вида **array[i]** се превръщат в ***(array+i)**. **Да обобщим: Езикът C++ осигурява два начина за обръщение към елементи на масиви - чрез индекси (удобно) и чрез указатели (бързо).**

Трябва да обърнем внимание, че **името на масив е константа**, **а съответният указател е променлива**.Затова израз като **str++** за горния масив е грешен. По-общо казано всеки израз, който би се опитал да промени **str** е грешен!

Ще разгледаме две версии на една и съща програма, с аритметични операции над указатели и с индексиране на масиви. И в двата случая един въведен от клавиатура текст се печати по една дума на ред. Програмистите наричат разграничените (в случай от интервал) символни последователности **лексеми** (**token**). Т.е. задачата може да се формулира така: да се намерят лексемите и се отпечатат на екрана.

Първото решение prog_44a е по-трудно за начинаещите програмисти. Обърнете внимание на следната инструкция: **if(*p) {p++;}**

Без нея низа не се чете докрая, а процесът „забива“ на първия интервал след първата прочетена дума.

```
#include <iostream> // nashata 44a programa
using namespace std;// primer za razbivane na edin red na dumi
int main() {
	char str[80], token[80];
	char *p, *q;
	cout << "Vavedete izrechenie: ";
	gets(str); p = str;
	while(*p) {
		q = token; // chetem dokato se sreshtne interval ili redyt svyrshi
		while(*p != ' ' && *p) {
			*q = *p; // kopirame simvola
			q++; p++; //otivame na sledvashtia simvol v token i str
```

```
        }
        if(*p) {p++;} // premestvame se sled tekushtia interval
        *q = '\0'; // slagame na leksemata-niz simvol za krai
        cout << token << '\n'; // t.e. moje i edin simvol vmesto string
    } return 0;}
#include <iostream> // nashata 44b programa
using namespace std;// primer za razbivane na edin red na dumi
int main() {// syshtata kato prog_44a no s indeksirani masivi vmesto ukazateli
    char str[80];
    char token[80];
    int i, k;
    cout << "Vavedete izrechenie: ";
    gets(str);
    for(i=0; str[i]; i++) {
        for(k=0; str[i]!=' ' && str[i]; k++, i++) { // spira pri interval ili kraia na niza
            token[k] = str[i];
        }
```

```
        token[k] = '\0';// slagame na leksemata-niz simvol za krai
        cout << token << '\n'; // t.e. moje i edin simvol vmesto string
    }
    return 0;
}
```

## Указатели и низове

Низовите константи могат да се използват да инициализират също и символен указател, както в показано в следващата програма:

```
#include <iostream> // nashata 45-ta programa
using namespace std;// primer za ukazateli i nizove
int main() {
    char *ps;// ili char *ps="Uchim se da programirame na C++!\n";
    ps = "Uchim se da programirame na C++!\n"; // ne mojem da go promeniame
    cout << ps;
//  ps[0] = 'u'; // ! dava greshka, ne mojem da promenim niza
    return 0;}
```

Компилаторът разполага низа в паметта и връща адреса на първия символ, т.е. на указателя се присвоява адреса на първия символ на низа. Низът е константа и не може да се променя, в което и сами можем да се убедим, ако премахнем коментара на предпоследния ред.

## Масив от указатели

Като повечето типове, указателите също могат да бъдат елементи на масив. Обобщената конструкция на дефиниране на масив от указатели е:

**тип *име_на_масив[размер];**

**Забележете, че име на масив от указатели се явява указател-към-указател.**

В следващата програма 46 косвено се присвояват стойности на променливите, сочени от указателите. Ако искаме да узнаем (за нашият компилатор) колко байта заема един указател може да използваме код подобен на следния:

```
    float *p;
    cout << "tipyt ukazatel zaema  " << sizeof(p)<< " baita\n";
```
Независимо от типът на обекта всички указатели имат еднаква дължина!
```
#include <iostream> // nashata 46-ta programa
using namespace std;// primer za masiv ot ukazateli
int main() {
    int x, y, z;
    int* arr_ptr[3];// masiv ot ukazateli
    int i;
    arr_ptr[0] = &x;
    arr_ptr[1] = &y;
    arr_ptr[2] = &z; // kosveno prisvoiavane v sledvashtia for
    for (i = 0 ; i < 3 ; i++) {  *arr_ptr[i] = i+10; }
    cout << "x=" << x << "\ny=" << y<< "\nz=" << z << '\n';
    return 0;
}
```

Най-често масив от указатели се използва с низове, с цел удобно построяване и използване на таблица от низове. Пример за това е програма_47.

```
#include <iostream> // nashata 47-a programa
using namespace std;// masiv ot ukazateli kym nizove
int main() {
char *StringTable[7] ={ "Ponedelnik", "Vtornik", "Sriada",
    "Chetvyrtyk", "Petyk", "Sybota", "Nedelia" };
    for(int index = 0 ; index < 7 ; index++) {
        cout <<  StringTable[index] << "\n" ;
    }// zabelejete kyde e definirana promenlivata index
    return 0;
}
```

## Обобщени указатели (към тип void)

Освен указателите от известните ни основни типове съществуват и т.нар. обобщени указатели. Това са указатели към тип **void**, затова се наричат още void-указатели.

**void *име_на_указател;** **Обобщените указатели могат да сочат към променливи от всякакви типове.**

Можем да присвояваме адресите на променливите пряко на void-указателя. Компилаторът преобразува автоматично адреса на променливата във void-указател. **Достъпът до обект чрез void-указател обаче изисква явно преобразуване на void-указателя в указател към типа на обекта**. Защото трябва да се знае колко байта трябва да прочетат/запишат. Казано с прости думи, това преобразуване инструктира компилатора да третира void указателя все едно е указател към обект от указания тип. Пример: програма48

```
#include <iostream> // nashata 48-a programa
using namespace std;// obobshteni ukazateli (void ukazateli)
int main() {
	void *pv;
	int i = 10;
	double d = 56.23;
	pv = &i; // neiavno preobrazuvane na tip  int* v tip void*
	cout << "*pv = " <<  *((int*)pv) << "\n"; // iavno preobrazuvane na tip  void*
v tip int*
	pv = &d;
// sledva iavno preobrazuvane na tip  void* v tip double*
// obyrnete vimanie na konstrukciata *((double*)pv)
	cout << "*pv = " <<  *((double*)pv) << "\n";
	return 0;
}
```

## Указател към указател (непряка адресация на две нива)

Възможно е да се  създаде указател, който да се насочи към друг указател, а вторият — към крайната стойност. Такава ситуация се нарича непряка (косвена) адресация на две нива или указател към указател. Т.е. първият указател съдържа адреса на втория, а вторият - адресът на обикновена променлива. Може да се организира конструкция и на повече от две нива.

**За да получи първият указател достъп към стойността на обикновената променлива е необходимо двукратно да се приложи оператор “*”**, както е показано в следващия кратък пример.

```
#include <iostream> // nashata 49-a programa
using namespace std;//  ukazatel kym ukazatel
int main() {
    int x, *p, **q;
    x = 12;
    p = &x;
    q = &p;
// izvejda se stoinostta na promenlivata x
// syshtoto prawi i cout << * p;
    cout << **q;
    return 0;
}
```

## Спецификаторът const и указатели

Спецификаторът **const** може да се прилага както към обект, сочен от указател, така и към самия указател или едновременно към обекта и указателя.

Ще разгледаме и трите възможни комбинации:

**А) const тип * име_на_указател;**

Това означава, че указателят не може да променя обекта. Самият указател може да се променя обаче, т.е. да се насочи към друг обект.

**Б) тип * const име_на_указател = &име_на_променлива;**

Това означава, че указателят може да променя обекта. Самият указател обаче не може да се променя (т.е константен-указател към променлива). Обърнете внимание, че в тоя случай указателят трябва да се инициализира с

адрес на променлива **по време на дефинирането си**. Използва се по-рядко от А).

**В) const тип * const име_на_указател = &име_на_променлива;**

Това означава, че указателят не може да променя обекта и самият указател също не може да се променя. Среща се най-рядко.

Последната, програма 50, показва употребата на спецификаторът **const** за първите два случая.

```
#include <iostream> // nashata 50-a programa
using namespace std; // specifikatoryt const i ukazateli
int main() {
	int x=10, y=7;
	const int *px;
	int* const p2x = &x;
	px = &x;
	cout <<"x = " << *px << "\n";
// *px = 20;  // !!! Greshka, obektyt ne moje da se promenia ot px
	px = &y; // pozvoleno
	cout <<"y = " << *px << "\n";
	*p2x = 20; // pozvoleno
	cout <<"x = " << *p2x << "\n";
//	p2x = &y; // !!! Greshka,  p2x ne moje da  se nasochi kym drug obekt
	return 0;
}
```