

# Семинар по „Увод в програмирането“

## Указатели

### 1. Указатели(Pointers)

#### 1) Какво представляват указателите?

Указателите са променливи, които пазят адрес. Този адрес обикновено е адрес в паметта, на който се намира някаква друга променлива. Указател от даден тип пази адреса на променлива от същия тип. Освен адреса на дадена променлива в паметта, един указател може да бъде зададен да не сочи към никъде, т.е. да сочи към `nullptr`(празно пространство).

#### 2) Дефиниция на указател

```
int data = 5;  
int* ptr = &data;
```

Дефиницията на указател се различава от дефиницията на стандартна променлива със символа \*, която се намира след типа на указателя. Синтаксиса на указателя позволява звездата да се намира и пред името на указателя. Как ще се изпише е решение изцяло на програмиста.

#### 3) Свойства на указателите

- Един указател може да бъде преместван да сочи към друго място
- Чрез указател можем да променим данните, които стоят зад този адрес
- Тъй като указателят също е обект, той притежава адрес също

Примери:

Пренасочване на пойнтьър:

```
int data = 2;  
int* ptr = &data;  
  
std::cout << "First value behind ptr: " << (*ptr) << std::endl;  
  
int number = 20;  
ptr = &number;  
  
std::cout << "New value behind ptr: " << (*ptr) << std::endl;
```

Резултат:

```
First value behind ptr: 2
New value behind ptr: 20
```

Промяна на данните зад указател:

```
int data = 2;
int* ptr = &data;

std::cout << "First value behind ptr: " << (*ptr) << std::endl;

(*ptr)++;

std::cout << "New value behind ptr: " << (*ptr) << std::endl;
```

Резултат:

```
First value behind ptr: 2
New value behind ptr: 3
```

Адрес на указател:

```
int data = 2;
int* ptr = &data;

std::cout << "Address of ptr: " << &ptr << std::endl;
```

Резултат:

```
Address of ptr: 003EF738
```

#### 4) Указател към константа/ Константен указател

Указателите към константа не позволяват да се променя стойността на променливата, чийто адрес пази указателя. Пример:

```
int firstNumber = 2;
int* firstPtr = &firstNumber;
(*firstPtr)--;
const int* secondPtr = &firstNumber;
(*secondPtr)--;
```

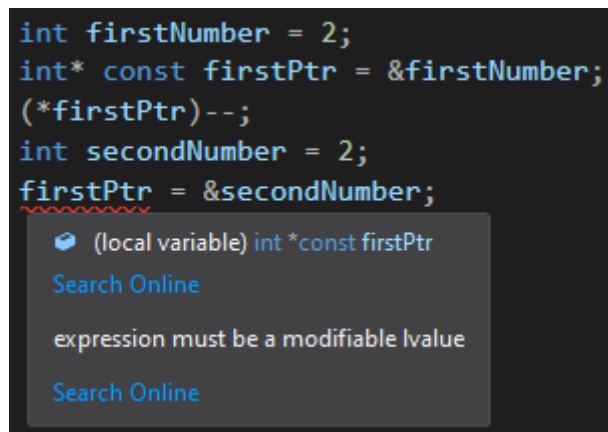
expression must be a modifiable lvalue

[Search Online](#)

Както се вижда на примера компилатора ни прави забележка. **secondPtr** е указател към константа и не можем да променим стойността зад този указател. За разлика от него **firstPtr** е обикновен указател и на него можем да му променим стойността.

Константните указатели са такива указатели, при които не можем да променяме адреса, който сочат. За разлика от **указателите към константа** можем да променяме стойността на променливата, към която сочат. Пример:

```
int firstNumber = 2;
int* const firstPtr = &firstNumber;
(*firstPtr)--;
int secondNumber = 2;
firstPtr = &secondNumber;
```



(local variable) int \*const firstPtr  
Search Online  
expression must be a modifiable lvalue  
Search Online

Тук компилатора отново ни подсказва. **firstPtr** е **константен указател** и не можем да променим адреса на променливата, към която сочи.

#### a. Pointer arithmetic

Тъй като указателите пазят адрес, към който сочат, този адрес може да бъде променен. Понеже един указател знае от какъв тип е, той знае точно колко байта да се измести в ляво или дясно, за да стигне до следващия елемент. За да се изпълни тази операция е нужно едно множество от елементи да бъде последователно разположено в паметта. Преместването на указателя се случва без да се повлияе на стойността, която той сочи преди това. Така се появява и Pointer arithmetic. Операторите, които са ни нужни, за **Pointer arithmetic** са ++, --, +, -, Пример:

Нека имаме масив от 5 на брой целочислени елементи . Можем да го обходим по следния начин:

```
int arr[5];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;
arr[4] = 5;

for (int i = 0; i < 5; i++)
{
    std::cout << (*arr + i) << " ";
}
```

Чрез `*(arr + i)` казваме на указателя с точно колко байта да се премести, за да достигем желаната от нас стойност.

Друг начин това да се случи е:

```
int arr[5];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;
arr[4] = 5;

int* ptr = arr;

for (int i = 0; i < 5; i++)
{
    std::cout << (*ptr++) << " ";
}
```

Тук имаме отделен пойнтьър, `ptr`, чрез който обхождаме масива. Използваме оператора `++`.

## 2. Референции

### 1) Какво представляват референциите?

Референциите представляват друго име на вече съществуваща променлива. Можем да си представим, че референциите са константни указатели.

### 2) Дефиниция на референция

```
int data = 5;
int& ref = data;
```

Тук `&` е част от типа. За разлика от примера по-нагоре в този файл, при дефиницията на указател `&` ни дава адреса на променливата, която стои вдясно от знака.

### 3) Свойства на референциите

- Веднъж дефинирана една референция, не можем да сменим променливата, към която сочи
- За разлика от това, обаче можем да променим стойността на променливата, която сочи, чрез референция
- При инициализиране на референцията не се копират данни. Това спестява много време и памет.

### 4) Референции във функции

Ако приемаме даден параметър на функция по референция, то направените промени по този параметър ще се отразят и извън тази функция. Иначе, ако не приемем този параметър по референция, ще се направи копие на тази променлива, което при базовите типове се случва сравнително бързо, и направените промени няма да се отразят извън тази функция.

Пример:

```
1  #include <iostream>
2
3  void increment(int data)
4  {
5      data++;
6  }
7
8  int main()
9  {
10     int number = 2;
11     increment(number);
12
13     std::cout << number;
14     return 0;
15 }
```

Съответно резултата е следния:

```
2
D:\C++\FirstProject\Debug\FirstProject.exe (process 20100) exited with code 0.
Press any key to close this window . . .
```

Нека сега видим същия пример, но да приемем **data** по референция.

```

1  #include <iostream>
2
3  void increment(int& data)
4  {
5      data++;
6  }
7
8  int main()
9  {
10     int number = 2;
11     increment(number);
12
13     std::cout << number;
14     return 0;
15 }

```

Сега резултата е следния:

```

3
D:\C++\FirstProject\Debug\FirstProject.exe (process 8344) exited with code 0.
Press any key to close this window . . .

```

Сега, когато вече приемаме нашия параметър по референция, промените, направени във функцията, са отразени и извън нея.

Когато имаме функция връщаща референция, ние връщаме цялата променлива. Затова трябва да сме сигурни, че когато връщаме дадена променлива трябва да сме сигурни, че тя ще продължи да съществува и след тялото на функцията.