

# Указатели и псевдоними

Трифон Трифонов

Увод в програмирането,  
спец. Компютърни науки, 1 поток,  
спец. Софтуерно инженерство,  
2016/17 г.

7–14 декември 2016 г.

# Тип указател

- **Множество от стойности:** всички възможни `lvalue` от даден тип и специалната стойност `NULL`.
- Интегрален **нечислов** тип
- Параметризиран тип: ако `T` е тип данни, то `T*` е тип “указател към елемент от тип `T`”
- Физическо представяне: цяло число, указващо адреса на указваната `lvalue` в паметта
- Стойностите от тип “указател” са с размера на машинната дума
  - 32 бита (4 байта) за 32-битови процесорни архитектури
  - 64 бита (8 байта) за 64-битови процесорни архитектури

# Операции с указатели

- рефериране (&<lvalue>)
- дерефериране (\*<указател>)
  - унарна операция!
- сравнение (==, !=, <, >, <=, >=)
- указателна аритметика (+, -, +=, -=, ++, --)
- извеждане (<<)
- няма въвеждане! (>>)

# Дефиниране на указателни променливи

`<тип> *<име> [ = <израз> ] { , *<име> [ = <израз> ] };`

**Примери:**

# Дефиниране на указателни променливи

```
<тип> *<име> [ = <израз> ] { , *<име> [ = <израз> ] };
```

Примери:

- `int *pi;`



# Дефиниране на указателни променливи

`<тип> *<име> [ = <израз> ] { , *<име> [ = <израз> ] };`

Примери:

- `int *pi;`
- `double *pd = NULL;`

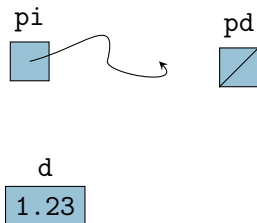


## Дефиниране на указателни променливи

```
<тип> *<име> [ = <израз> ] { , *<име> [ = <израз> ] };
```

## Примери:

- `int *pi;`
- `double *pd = NULL;`
- `double d = 1.23;`

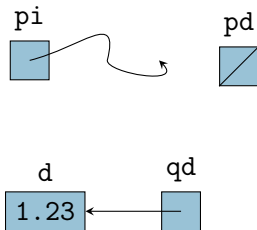


# Дефиниране на указателни променливи

```
<тип> *<име> [ = <израз> ] { , *<име> [ = <израз> ] };
```

## Примери:

- `int *pi;`
- `double *pd = NULL;`
- `double d = 1.23;`
- `double *qd = &d;`



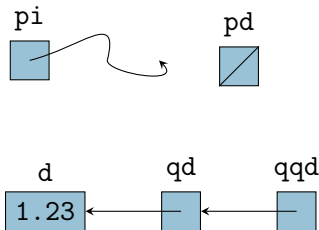


# Дефиниране на указателни променливи

```
<тип> *<име> [ = <израз> ] { , *<име> [ = <израз> ] };
```

## Примери:

- `int *pi;`
- `double *pd = NULL;`
- `double d = 1.23;`
- `double *qd = &d;`
- `double **qqd = &qd;`



# Рефериране и дерефериране

- `&<име>` — указател към променливата `<име>`

## Рефериране и дерефериране

- `&<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`

## Рефериране и дерефериране

- `&<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`

# Рефериране и дерефериране

- `&<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`



# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`
  - `&x = p;`

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`
  - `&x = p;`
- `*<rvalue>` връща като резултат `<lvalue>!`

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`
  - ~~`&x = p;`~~
- `*<rvalue>` връща като резултат `<lvalue>!`
  - `*p = x;`

# Рефериране и дерефериране

- `<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`
  - ~~`&x = p;`~~
- `*<rvalue>` връща като резултат `<lvalue>!`
  - `*p = x;`
  - `**qqd = 3.15;`

# Рефериране и дерефериране

- `&<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`
  - ~~`&x = p;`~~
- `*<rvalue>` връща като резултат `<lvalue>!`
  - `*p = x;`
  - `**qqd = 3.15;`
- операциите са дуални една на друга и се унищожават взаимно

# Рефериране и дерефериране

- `&<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`
  - ~~`&x = p;`~~
- `*<rvalue>` връща като резултат `<lvalue>!`
  - `*p = x;`
  - `**qqd = 3.15;`
- операциите са дуални една на друга и се унищожават взаимно
  - $\&(*p) \iff p$



# Рефериране и дерефериране

- `&<име>` — указател към променливата `<име>`
- `*<указател>` — мястото в паметта, сочено от `<указател>`
- **Примери:**
  - `int x = 5, *p = &x;`
  - `int *q = p, y = *p + 2;`
  - `*p++; p = &y;`
  - `*q = 1; *p = *q;`
- `&<lvalue>` връща като резултат `<rvalue>!`
  - `&3`
  - ~~`&x = p;`~~
- `*<rvalue>` връща като резултат `<lvalue>!`
  - `*p = x;`
  - `**qqd = 3.15;`
- операциите са дуални една на друга и се унищожават взаимно
  - `&(*p)  $\iff$  p`
  - `*(&x)  $\iff$  x`

# Указатели и масиви

В C++ има много тясна връзка между указатели и масиви.

## Факт

Името на масив е **константен указател** към първия му елемент.

# Указатели и масиви

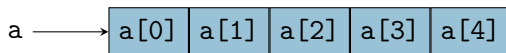
В C++ има много тясна връзка между указатели и масиви.

## Факт

Името на масив е **константен указател** към първия му елемент.

## Примери:

- `int a[5];`



# Указатели и масиви

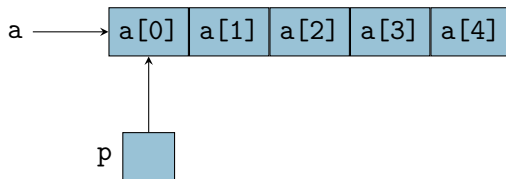
В C++ има много тясна връзка между указатели и масиви.

## Факт

Името на масив е **константен указател** към първия му елемент.

## Примери:

- `int a[5];`
- `int* p = a;`



# Указатели и масиви

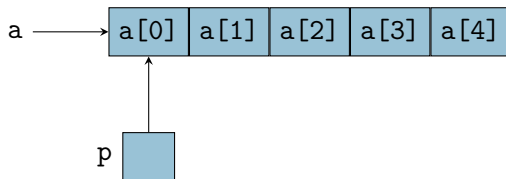
В C++ има много тясна връзка между указатели и масиви.

## Факт

Името на масив е **константен указател** към първия му елемент.

## Примери:

- `int a[5];`
- `int* p = a;`
- `*p = 15;`



# Указатели и масиви

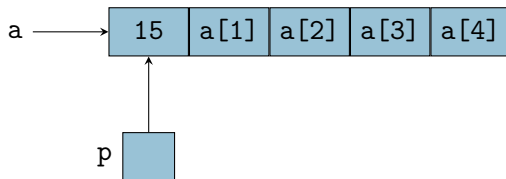
В C++ има много тясна връзка между указатели и масиви.

## Факт

Името на масив е **константен указател** към първия му елемент.

## Примери:

- `int a[5];`
- `int* p = a;`
- `*p = 15;`
- `cout << a[0];`



# Указатели и масиви

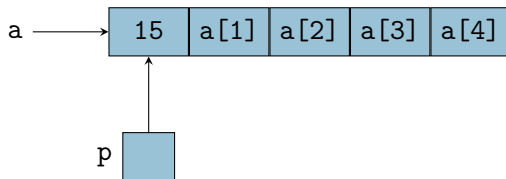
В C++ има много тясна връзка между указатели и масиви.

## Факт

Името на масив е **константен указател** към първия му елемент.

## Примери:

- `int a[5];`
- `int* p = a;`
- `*p = 15;`
- `cout << a[0];`
- `*a = 20; a = p;`



# Указатели и масиви

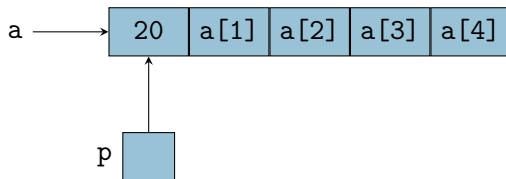
В C++ има много тясна връзка между указатели и масиви.

## Факт

Името на масив е **константен указател** към първия му елемент.

## Примери:

- `int a[5];`
- `int* p = a;`
- `*p = 15;`
- `cout << a[0];`
- `*a = 20; a = p;`





# Указателна аритметика

- Указателната аритметика позволява по дадена отправна точка в паметта (указател) да реферираме съседни на нея клетки.

# Указателна аритметика

- Указателната аритметика позволява по дадена отправна точка в паметта (указател) да реферираме съседни на нея клетки.
- За целта трябва да укажем колко клетки напред или назад в паметта искаме да прескочим.

# Указателна аритметика

- Указателната аритметика позволява по дадена отправна точка в паметта (указател) да реферираме съседни на нея клетки.
- За целта трябва да укажем колко клетки напред или назад в паметта искаме да прескочим.
- Синтаксис:

# Указателна аритметика

- Указателната аритметика позволява по дадена отправна точка в паметта (указател) да реферираме съседни на нея клетки.
- За целта трябва да укажем колко клетки напред или назад в паметта искаме да прескочим.
- Синтаксис:
  - `<указател> [+ | -] <цяло_число>`

# Указателна аритметика

- Указателната аритметика позволява по дадена отправна точка в паметта (указател) да реферираме съседни на нея клетки.
- За целта трябва да укажем колко клетки напред или назад в паметта искаме да прескочим.
- Синтаксис:
  - `<указател> [+ | -] <цяло_число>`
  - `<цяло_число> + <указател>`

# Указателна аритметика

- Указателната аритметика позволява по дадена отправна точка в паметта (указател) да реферираме съседни на нея клетки.
- За целта трябва да укажем колко клетки напред или назад в паметта искаме да прескочим.
- Синтаксис:
  - $\langle \text{указател} \rangle [+ \mid -] \langle \text{цяло\_число} \rangle$
  - $\langle \text{цяло\_число} \rangle + \langle \text{указател} \rangle$
- прескачаме  $\langle \text{цяло\_число} \rangle$  клетки напред (+) или назад (-) от адреса, сочен от  $\langle \text{указател} \rangle$

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- Зависи от типа, който указваме!



## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- Зависи от типа, който указваме!
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- **Зависи от типа, който указваме!**
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;
  - $p + 2$  означава “прескочи 8 байта напред”, ако `int* p`;

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- **Зависи от типа, който указваме!**
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;
  - $p + 2$  означава “прескочи 8 байта напред”, ако `int* p`;
  - $p + 2$  означава “прескочи 16 байта напред”, ако `double* p`;

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- **Зависи от типа, който указваме!**
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;
  - $p + 2$  означава “прескочи 8 байта напред”, ако `int* p`;
  - $p + 2$  означава “прескочи 16 байта напред”, ако `double* p`;
- `sizeof(<тип>|<израз>)` — размера в байтове, заемаан в паметта от <израз> или от стойност от <тип>

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- **Зависи от типа, който указваме!**
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;
  - $p + 2$  означава “прескочи 8 байта напред”, ако `int* p`;
  - $p + 2$  означава “прескочи 16 байта напред”, ако `double* p`;
- `sizeof(<тип>|<израз>)` — размера в байтове, заемаан в паметта от <израз> или от стойност от <тип>
- Така, ако имаме `T* p`;

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- **Зависи от типа, който указваме!**
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;
  - $p + 2$  означава “прескочи 8 байта напред”, ако `int* p`;
  - $p + 2$  означава “прескочи 16 байта напред”, ако `double* p`;
- `sizeof(<тип>|<израз>)` — размера в байтове, заеман в паметта от <израз> или от стойност от <тип>
- Така, ако имаме `T* p`;
- ...тогава  $p + i$  прескача  $i * \text{sizeof}(T)$  байта напред

## Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- **Зависи от типа, който указваме!**
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;
  - $p + 2$  означава “прескочи 8 байта напред”, ако `int* p`;
  - $p + 2$  означава “прескочи 16 байта напред”, ако `double* p`;
- `sizeof(<тип>|<израз>)` — размера в байтове, заеман в паметта от <израз> или от стойност от <тип>
- Така, ако имаме `T* p`;
- ...тогава  $p + i$  прескача  $i * \text{sizeof}(T)$  байта напред
- `(int)p` — цялото число, съответстващо на адреса сочен от  $p$

# Големина на тип

- Но... какво означава “прескачаме  $n$  клетки”?
- **Зависи от типа, който указваме!**
  - $p + 2$  означава “прескочи 2 байта напред”, ако `char* p`;
  - $p + 2$  означава “прескочи 8 байта напред”, ако `int* p`;
  - $p + 2$  означава “прескочи 16 байта напред”, ако `double* p`;
- `sizeof(<тип>|<израз>)` — размера в байтове, заемаан в паметта от <израз> или от стойност от <тип>
- Така, ако имаме `T* p`;
- ...тогава  $p + i$  прескача  $i * \text{sizeof}(T)$  байта напред
- `(int)p` — цялото число, съответстващо на адреса сочен от `p`
- $p + i \iff (T*)((int)p + i * \text{sizeof}(T))$



# Указателна аритметика за масиви

## Факт

Името на масив е **константен указател** към първия му елемент.

# Указателна аритметика за масиви

## Факт

Името на масив е **константен указател** към първия му елемент.

Освен това,  $a[i] \iff *(a + i)$

# Указателна аритметика за масиви

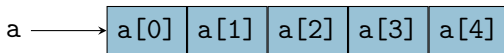
## Факт

Името на масив е **константен указател** към първия му елемент.

Освен това,  $a[i] \iff *(a + i)$

## Примери:

- `int a[5], x;`



# Указателна аритметика за масиви

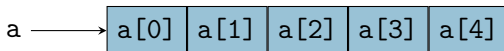
## Факт

Името на масив е **константен указател** към първия му елемент.

Освен това,  $a[i] \iff *(a + i)$

## Примери:

- `int a[5], x;`
- `cout << *a;`



# Указателна аритметика за масиви

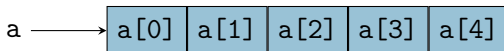
## Факт

Името на масив е **константен указател** към първия му елемент.

Освен това,  $a[i] \iff *(a + i)$

## Примери:

- `int a[5], x;`
- `cout << *a;`
- `*(a + 1) = 7;`



# Указателна аритметика за масиви

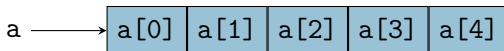
## Факт

Името на масив е **константен указател** към първия му елемент.

Освен това,  $a[i] \iff *(a + i)$

## Примери:

- `int a[5], x;`
- `cout << *a;`
- `*(a + 1) = 7;`
- `*(a + 4)--;`



# Указателна аритметика за масиви

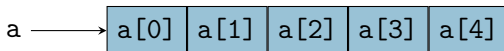
## Факт

Името на масив е **константен указател** към първия му елемент.

Освен това,  $a[i] \iff *(a + i)$

## Примери:

- `int a[5], x;`
- `cout << *a;`
- `*(a + 1) = 7;`
- `*(a + 4)--;`
- ~~`a++; a--; a = &x;`~~



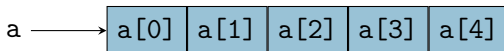
# Указателна аритметика за масиви

## Факт

Името на масив е **константен указател** към първия му елемент.

Освен това,  $a[i] \iff *(a + i)$

## Примери:

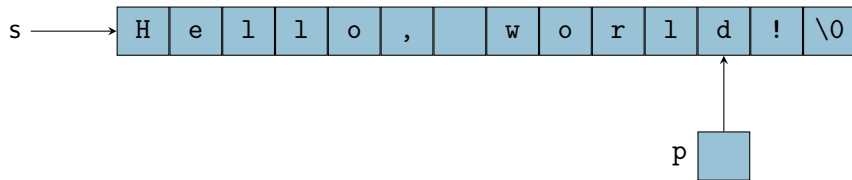


- `int a[5], x;`
- `cout << *a;`
- `*(a + 1) = 7;`
- `*(a + 4)--;`
- ~~`a++; a--; a = &x;`~~
- Странно, но вярно:  $a[i] \iff *(a+i) \iff *(i+a) \iff i[a]$



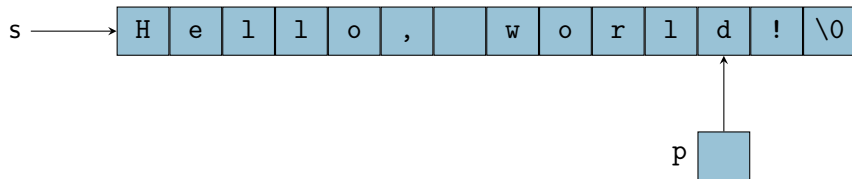
## Указатели и низове

Низовете са масиви от символи



# Указатели и низове

Низовете са масиви от символи

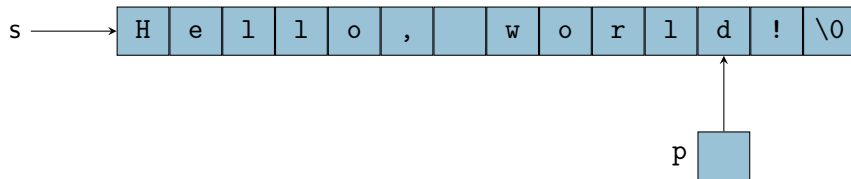


Примери:

```
char s[] = "Hello, world!";
```

# Указатели и низове

Низовете са масиви от символи



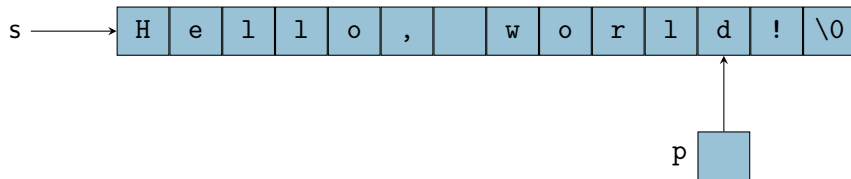
Примери:

```
char s[] = "Hello, world!";
```

```
void print(char* p) {  
    while (*p) cout << *p++;  
}
```

# Указатели и низове

Низовете са масиви от символи



Примери:

```
char s[] = "Hello, world!";
```

```
void print(char* p) {
    while (*p) cout << *p++;
}
```

```
int strlen(char* p) {
    int n = 0;
    while (*p++) n++;
    return n;
}
```

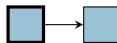
# Указатели и константи

- Константен указател (който е константа)

# Указатели и константи

- Константен указател (който е константа)

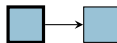
- `<тип>* const`



# Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`
- `int x, *p = &x;`



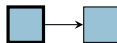
## Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`

- `int x, *p = &x;`

- `int* const q = p; q = p + 2; *q = 5;`





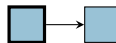
# Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`

- `int x, *p = &x;`

- `int* const q = p; q = p + 2; *q = 5;`



- Указател към константа (сочещ към константа)

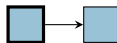
# Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`

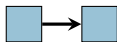
- `int x, *p = &x;`

- `int* const q = p; q = p + 2; *q = 5;`



- Указател към константа (сочещ към константа)

- `const <тип>*  $\iff$  <тип> const*`



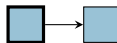
# Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`

- `int x, *p = &x;`

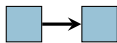
- `int* const q = p; q = p + 2; *q = 5;`



- Указател към константа (сочещ към константа)

- `const <тип>* ⇔ <тип> const*`

- `int x, *p = &x;`



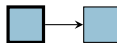
## Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`

- `int x, *p = &x;`

- `int* const q = p; q = p + 2; *q = 5;`

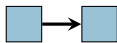


- Указател към константа (сочещ към константа)

- `const <тип>* ⇔ <тип> const*`

- `int x, *p = &x;`

- `int const* q = &x; q++; p = q; *q = 5;`



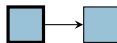
# Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`

- `int x, *p = &x;`

- `int* const q = p; q = p + 2; *q = 5;`

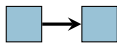


- Указател към константа (сочещ към константа)

- `const <тип>* ⇔ <тип> const*`

- `int x, *p = &x;`

- `int const* q = &x; q++; p = q; *q = 5;`



- Ако `p` е указател към константа, то `*p` е `<rvalue>`

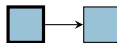
# Указатели и константи

- Константен указател (който е константа)

- `<тип>* const`

- `int x, *p = &x;`

- `int* const q = p; q = p + 2; *q = 5;`

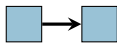


- Указател към константа (сочещ към константа)

- `const <тип>* ⇔ <тип> const*`

- `int x, *p = &x;`

- `int const* q = &x; q++; p = q; *q = 5;`



- Ако `p` е указател към константа, то `*p` е `<rvalue>`

- Ако `x` е константа, то `&x` е указател към константа

# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).

# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).  
Низовите константи са **указатели към константен символ** (`char const*`).

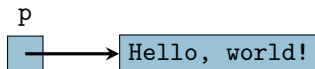


# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).  
Низовите константи са **указатели към константен символ** (`char const*`).

- `char const* p = "Hello, world!";`



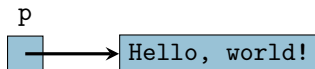
# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).  
Низовите константи са **указатели към константен символ** (`char const*`).

- `char const* p = "Hello, world!";`

- ~~`char* q = "Hi C++!";`~~

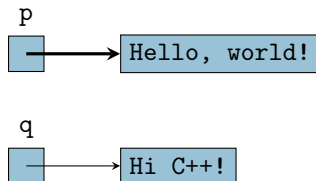


# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).  
 Низовите константи са **указатели към константен символ** (`char const*`).

- `char const* p = "Hello, world!";`
- ~~`char* q = "Hi C++!";`~~
- `char q[] = "Hi C++!";`

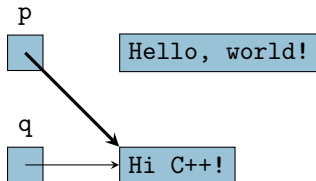


# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).  
 Низовите константи са **указатели към константен символ** (`char const*`).

- `char const* p = "Hello, world!";`
- ~~`char* q = "Hi C++!";`~~
- `char q[] = "Hi C++!";`
- `p = q;`

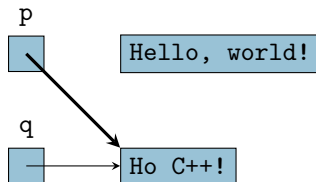


# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).  
 Низовите константи са **указатели към константен символ** (`char const*`).

- `char const* p = "Hello, world!";`
- ~~`char* q = "Hi C++!";`~~
- `char q[] = "Hi C++!";`
- `p = q;`
- `q[1] = 'o';` ~~`p[1] = 'o';`~~

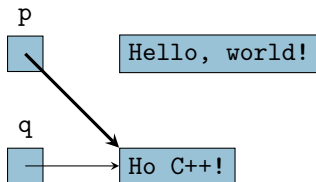


# Указатели и низови константи

## Факт

Името на низ е **константен указател** към първия му символ (`char* const`).  
 Низовите константи са **указатели към константен символ** (`char const*`).

- `char const* p = "Hello, world!";`
- ~~`char* q = "Hi C++!";`~~
- `char q[] = "Hi C++!";`
- `p = q;`
- `q[1] = 'o';` ~~`p[1] = 'o';`~~
- `cout << p[4];`



## Указатели и многомерни масиви

- `int a[2][2][3];`

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]

# Указатели и многомерни масиви

- `int a[2][2][3];`
- `a` е от тип `int*const[2][3];`

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]



# Указатели и многомерни масиви

- `int a[2][2][3];`
- `a` е от тип `int*const[2][3];`
- `a[i]` е от тип `int*const[3];`

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]

# Указатели и многомерни масиви

- `int a[2][2][3];`
- `a` е от тип `int*const[2][3];`
- `a[i]` е от тип `int*const[3];`
- `a[i][j]` е от тип `int*const;`

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]

## Указатели и многомерни масиви

- `int a[2][2][3];`                      • `a[i]  $\iff$  *(a+i)`
- `a` е от тип `int*const[2][3];`
- `a[i]` е от тип `int*const[3];`
- `a[i][j]` е от тип `int*const;`

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]

## Указатели и многомерни масиви

- `int a[2][2][3];`
- `a` е от тип `int*const[2][3];`
- `a[i]` е от тип `int*const[3];`
- `a[i][j]` е от тип `int*const;`
- $a[i] \iff *(a+i)$
- $a[i][j] \iff (*(a+i)+j)$

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]

## Указатели и многомерни масиви

- `int a[2][2][3];`
- `a` е от тип `int*const[2][3];`
- `a[i]` е от тип `int*const[3];`
- `a[i][j]` е от тип `int*const;`
- $a[i] \iff *(a+i)$
- $a[i][j] \iff (*(a+i)+j)$
- $a[i][j][k] \iff (*(a+i)+j)+k)$

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]

## Указатели и многомерни масиви

- `int a[2][2][3];`
- `a` е от тип `int*const[2][3];`
- `a[i]` е от тип `int*const[3];`
- `a[i][j]` е от тип `int*const;`
- $a[i] \iff *(a+i)$
- $a[i][j] \iff (*(a+i)+j)$
- $a[i][j][k] \iff (*(a+i)+j)+k)$
- $a[1][1][1] \iff (*(a+1)+1)+1)$

a											
a[0]						a[1]					
a[0][0]			a[0][1]			a[1][0]			a[1][1]		
a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип



## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`
- × **Няма** автоматично преобразуване от `void*` към `T*`

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`
- × **Няма** автоматично преобразуване от `void*` към `T*`
  - `int* p; void* q = p;`

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`
- × **Няма** автоматично преобразуване от `void*` към `T*`
  - `int* p; void* q = p;`
  - ~~`int* r = q;`~~

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`
- × **Няма** автоматично преобразуване от `void*` към `T*`
  - `int* p; void* q = p;`
  - ~~`int* r = q;`~~
  - `int* s = (int*)q;`

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`
- × **Няма** автоматично преобразуване от `void*` към `T*`
  - `int* p; void* q = p;`
  - ~~`int* r = q;`~~
  - `int* s = (int*)q;`
- × **Няма** дереферирание (`void` е празният тип)



## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`
- × **Няма** автоматично преобразуване от `void*` към `T*`
  - `int* p; void* q = p;`
  - ~~`int* r = q;`~~
  - `int* s = (int*)q;`
- × **Няма** дереферирание (`void` е празният тип)
  - `int x; void* p = &x;`

## Указател към неизвестен тип

- **Проблем:** Не можем да насочваме един и същ указател към променливи от различен тип!
- **Решение:** `void*` — указател към неизвестен тип
- ✓ Преобразуваме автоматично от `T*` към `void*`
  - `int x, *p; void *q = p;`
  - `void *r = &x, *pr = &r, *s = &r;`
- × **Няма** автоматично преобразуване от `void*` към `T*`
  - `int* p; void* q = p;`
  - ~~`int* r = q;`~~
  - `int* s = (int*)q;`
- × **Няма** дереферирание (`void` е празният тип)
  - `int x; void* p = &x;`
  - ~~`*p = 2; void y = *p;`~~

# Псевдоним

- **Множество от стойности:** всички възможни lvalue от даден тип

# Псевдоним

- **Множество от стойности:** всички възможни lvalue от даден тип
- **Параметризиран тип:** ако  $T$  е тип данни, то  $T\&$  е тип “псевдоним на елемент от тип  $T$ ”

# Псевдоним

- **Множество от стойности:** всички възможни lvalue от даден тип
- Параметризиран тип: ако  $T$  е тип данни, то  $T\&$  е тип “псевдоним на елемент от тип  $T$ ”
- Физическо представяне:

# Псевдоним

- **Множество от стойности:** всички възможни lvalue от даден тип
- Параметризиран тип: ако  $T$  е тип данни, то  $T\&$  е тип “псевдоним на елемент от тип  $T$ ”
- Физическо представяне:
  - на теория: както реши компилаторът

# Псевдоним

- **Множество от стойности:** всички възможни lvalue от даден тип
- Параметризиран тип: ако  $T$  е тип данни, то  $T\&$  е тип “псевдоним на елемент от тип  $T$ ”
- Физическо представяне:
  - на теория: както реши компилаторът
  - на практика: екивалентно на **константен указател към  $T$**

# Псевдоним

- **Множество от стойности:** всички възможни lvalue от даден тип
- Параметризиран тип: ако  $T$  е тип данни, то  $T\&$  е тип “псевдоним на елемент от тип  $T$ ”
- Физическо представяне:
  - на теория: както реши компилаторът
  - на практика: еквивалентно на **константен указател към  $T$**
  - $T\& \iff T* \text{ const}$



# Дефиниране на псевдоним

- $\langle \text{тип} \rangle \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle$   
 $\{, \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle \};$

# Дефиниране на псевдоним

- $\langle \text{тип} \rangle \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle$   
 $\{, \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle \};$
- инициализацията е **задължителна!**

# Дефиниране на псевдоним

- $\langle \text{тип} \rangle \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle$   
 $\{, \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle \};$
- инициализацията е **задължителна!**
  - както и на константните указатели

## Дефиниране на псевдоним

- $\langle \text{тип} \rangle \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle$   
 $\{ , \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle \};$
- инициализацията е **задължителна!**
  - както и на константните указатели
- псевдонимът **не може** да се пренасочва към друг обект

# Дефиниране на псевдоним

- $\langle \text{тип} \rangle \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle$   
 $\{, \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle \};$
- инициализацията е **задължителна!**
  - както и на константните указатели
- псевдонимът **не може** да се пренасочва към друг обект
  - както и константните указатели

# Дефиниране на псевдоним

- $\langle \text{тип} \rangle \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle$   
 $\{ , \& \langle \text{идентификатор} \rangle = \langle \text{обект} \rangle \};$
- инициализацията е **задължителна!**
  - както и на константните указатели
- псевдонимът **не може** да се пренасочва към друг обект
  - както и константните указатели

**Пример:**

# Дефиниране на псевдоним

- `<тип>& <идентификатор> = <обект>`  
`{, &<идентификатор> = <обект> };`
- инициализацията е **задължителна!**
  - както и на константните указатели
- псевдонимът **не може** да се пренасочва към друг обект
  - както и константните указатели

## Пример:

- `int x = 3;`

x

3

# Дефиниране на псевдоним

- `<тип>& <идентификатор> = <обект>`  
`{, &<идентификатор> = <обект> };`
- инициализацията е **задължителна!**
  - както и на константните указатели
- псевдонимът **не може** да се пренасочва към друг обект
  - както и константните указатели

## Пример:

- `int x = 3;`
- `int &a = x, b = a;`

x, a

3

b

3







# Дефиниране на псевдоним

- `<тип> &<идентификатор> = <обект>`  
`{, &<идентификатор> = <обект> };`
- инициализацията е **задължителна!**
  - както и на константните указатели
- псевдонимът **не може** да се пренасочва към друг обект
  - както и константните указатели

## Пример:

- `int x = 3;`
- `int &a = x, b = a;`
- `int &c = b;`
- `a = c + 5;`

x, a

8

b, c

3

## Сравнение на псевдоними и указатели

- Самият псевдоним не е обект, който може да бъде манипулиран

## Сравнение на псевдоними и указатели

- Самият псевдоним не е обект, който може да бъде манипулиран
  - Указателите могат да бъдат насочвани към различни адреси

## Сравнение на псевдоними и указатели

- Самият псевдоним не е обект, който може да бъде манипулиран
  - Указателите могат да бъдат насочвани към различни адреси
- Псевдонимите винаги са закачени за съществуващ обект

# Сравнение на псевдоними и указатели

- Самият псевдоним не е обект, който може да бъде манипулиран
  - Указателите могат да бъдат насочвани към различни адреси
- Псевдонимите винаги са закачени за съществуващ обект
  - Указателите могат да имат стойност `NULL`

## Сравнение на псевдоними и указатели

- Самият псевдоним не е обект, който може да бъде манипулиран
  - Указателите могат да бъдат насочвани към различни адреси
- Псевдонимите винаги са закачени за съществуващ обект
  - Указателите могат да имат стойност `NULL`
- Псевдонимът не се различава от оригинала



## Сравнение на псевдоними и указатели

- Самият псевдоним не е обект, който може да бъде манипулиран
  - Указателите могат да бъдат насочвани към различни адреси
- Псевдонимите винаги са закачени за съществуващ обект
  - Указателите могат да имат стойност NULL
- Псевдонимът не се различава от оригинала
  - Указателят изисква изрично дереферирание с \*

# Сравнение на псевдоними и указатели

- Самият псевдоним не е обект, който може да бъде манипулиран
  - Указателите могат да бъдат насочвани към различни адреси
- Псевдонимите винаги са закачени за съществуващ обект
  - Указателите могат да имат стойност NULL
- Псевдонимът не се различава от оригинала
  - Указателят изисква изрично дереферирание с \*
- Псевдонимите на един и същ обект са взаимнозаменяеми

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

**Пример:**

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

## Пример:

- `int a = 3;`

a  
3

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

## Пример:

- `int a = 3;`
- `a++;`

a  
4



# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

## Пример:

- `int a = 3;` a, b
- `a++;` 4
- `int& b = a;`

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

## Пример:

- `int a = 3;`
  - `a++;`
  - `int& b = a;`
  - `b++;`
- a, b
- 5

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

## Пример:

- `int a = 3;`
  - `a++;`
  - `int& b = a;`
  - `b++;`
  - `int const& c = b;`
- a, b, c
- 5

# Константни псевдоними

- `const <тип>&`  $\iff$  `<тип> const&`
- Представяват константен “изглед” на дадено място в паметта
- Няма разлика между константи псевдоними и псевдоними на константи

## Пример:

- `int a = 3;`

a, b, c

- `a++;`

5

- `int& b = a;`

- `b++;`

- `int const& c = b;`

- ~~`c++;`~~