

# Основы алгоритмизации и программирования

Лекция 8  
Указатели

# Определение указателей

**Указатель** – это переменная, которая может содержать адрес некоторого объекта

Формат  
декларации  
указателя

**тип \***

**ID\_указателя;**

Пример

р  
int \*a; double \*f; char \*w;

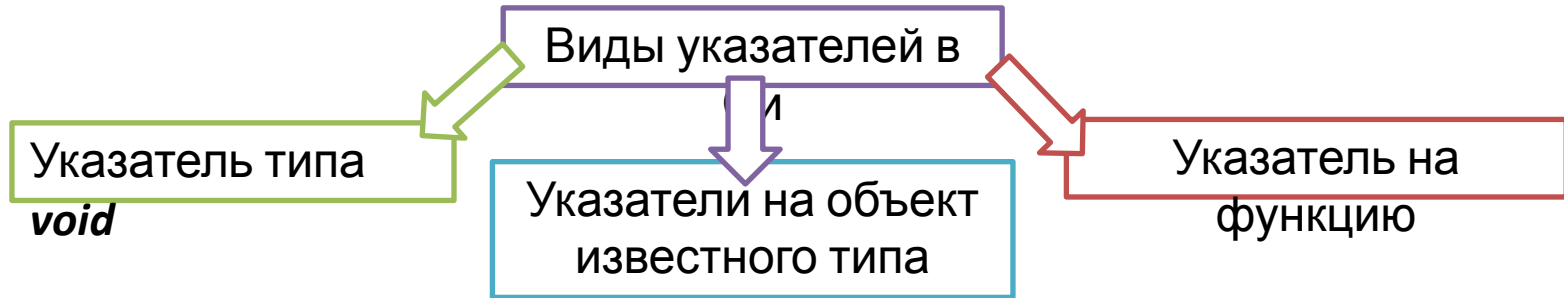
**Тип** может быть **любым**, кроме ссылки или битового поля, причем **тип** может быть к этому моменту только декларирован, но еще не определен

Символ «**звездочка**» не относится непосредственно к **ID указателя**, но для того, чтобы декларировать несколько указателей, ее нужно записывать перед именем каждого из них.

При обработке декларации любой переменной, например **double x=1.5**; компилятор выделяет для переменной **участок памяти**, размер которого определяется ее типом (**double** – 8 байт), и инициализирует его указанным значением (если таковое имеется). Далее все **обращения** в программе к **переменной** по имени **заменяются** компилятором на **адрес участка памяти**, в котором будет храниться значение этой переменной. Разработчик программы на языке Си имеет возможность определить собственные переменные для хранения адресов участков оперативной памяти. Такие переменные называются

# Определение указателей

**Значение указателя** равно первому байту участка памяти, на который он ссылается.



**Указатель не является самостоятельным типом данных**, так как всегда связан с каким-либо конкретным типом, т.е. **указатель** на объект **содержит адрес области памяти**, в которой хранятся данные определенного типа.

Указатель типа *void* применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, **не определен** (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов)

Указателю типа *void* можно присвоить значение указателя любого типа, а также сравнивать его с любыми другими указателями, но перед выполнением каких-либо действий с участком памяти, на которую он ссылается, **требуется явно преобразовать его к конкретному типу**

# Определение указателей

Указатель может быть константой или переменной, а также указывать на константу или переменную

С **указателями-переменными** связаны две унарные операции **&** и **\***.

Операция **\*** имеет смысл – «**значение, расположенное по указанному адресу**»

Операция **разадресации**, или **разыменования**, предназначена для **доступа к величине**, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины (если она не объявлена как константа)

**ID\_указателя = \*ID\_объекта;** – операция разыменования;

**\*ID\_указателя** – операция косвенной адресации

Операция **&** означает «**взять адрес**» операнда.

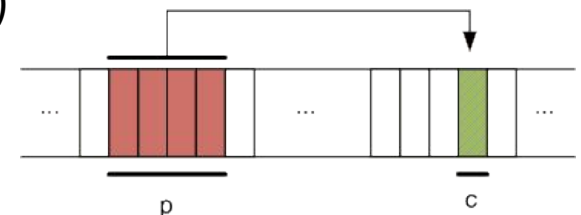
**Обращение к объектам** любого типа как операндам операций в языке Си

**по имени**

(идентификатор *nv*)

**по указателю**  
(операция  
адресации)

**косвенной**



```
char c; // переменная
char *p; // указатель
p = &c; // p = адрес c
```

# Определение указателей

Унарная операция получения адреса **&** применима к переменным, имеющим имя (*ID*), для которых **выделены участки оперативной памяти**. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной.

Отказ от именования объектов при наличии возможности доступа по указателю **приближает** язык **Си** по гибкости отображения «**объект – память**» к языку **ассемблера**

Приме

<sup>p</sup>  
`int x;` – переменная типа *int* ;  
`*y;` – указатель на объект типа *int*;  
`y = &x;` – *y* – адрес переменной *x*;  
`*y=1;` – косвенная адресация  
указателем поля *x*, т.е. по указанному адресу записать 1: *x* = 1.

Приме  
р

```
int i, j = 8, k = 5, *y;  
y=&i;  
*y=2;    – i = 2  
y=&j;  
*y+=i;   – j += i → j = j+i → j = j + 2 = 10  
y=&k;  
k+=*y;   – k += k → k = k + k = 10  
(*y)++;  – k++ → k = k + 1 = 10 + 1 = 11
```

Как видно из приведенных примеров, конструкцию ***\*ID\_указателя*** можно использовать в левой части оператора присваивания, так как она определяет адрес участка памяти. Эту конструкцию часто считают именем переменной, на которую ссылается указатель.

# Определение указателей

Пример

```
int i1;    – целая переменная;  
const int i2=1; – целая константа;  
int * pi1; – указатель на целую переменную;  
const int * pi2;    – указатель на целую константу;  
int * const pi1=&i1;    – указатель-константа на целую переменную;  
const int * const pi2=&i2; – указатель-константа на целую константу.
```

Модификатор **const**, находящийся между **ID** указателя и символом «звездочка», относится к самому указателю и запрещает его изменение, а **const** слева от звездочки задает константное значение объекта, на который он указывает. Для инициализации указателей использована операция получения адреса &.

Указатель подчиняется общим правилам определения области действия, видимости и времени жизни.



# Операция sizeof

Данная операция позволяет **определить размер** указанного параметра в байтах (тип результата *int*)

**sizeof ( *параметр* );**

***параметр*** – тип или идентификатор объекта (но не ID функции)

Если указан идентификатор **сложного объекта** (массив, структура, объединение), то **результатом** будет **размер всего объекта**

Приме

**sizeof(int)** – результат 2(4) байт;  
**double b[5];**  
**sizeof(b)** – результат 8 байт \* 5 = 40 байт.

Динамическое распределение **оперативной памяти** (ОП) связано с операциями **порождения** и **уничтожения** объектов по запросу программы, при котором захват и освобождение памяти производится **программно**, т.е. в процессе работы программы. При этом в языке **Си** порождение объектов (захват памяти) и уничтожение объектов (освобождение памяти) выполняются при помощи библиотечных функций

# Инициализация указателей

При **декларации указателя** желательно выполнить его **инициализацию**, т.е. присвоение начального значения. Наиболее распространенная из **ошибок** в программах – **непреднамеренное использование неинициализированных указателей**. **Инициализатор** записывается после **ID** указателя либо в круглых скобках, либо после знака равенства.

## Способы инициализации указателя

Присваивание указателю адреса существующего объекта

Использовать операцию получения адреса переменной

```
int a = 5;
```

```
int *p = &a;    – указателю p присвоили адрес объекта a;
```

```
int *p(&a);    – то же самое другим способом;
```

С помощью значения другого инициализированного указателя

```
int *g = p;
```

**Указателю-переменной** можно присвоить значение другого указателя либо выражения типа указатель.

С помощью идентификаторов массива или функции, которые трактуются как адрес начала участка памяти, в котором размещается указанный объект.

Причем, **ID** массивов и функций являются константными указателями. Такую константу можно присвоить переменной типа указатель, но нельзя подвергать преобразованиям:

```
int x[100], *y;
```

```
y = x;    – присваивание константы переменной;
```

```
x = y;    – ошибка, т.к. в левой части указатель-константа.
```



# Инициализация указателей

## Способы инициализации указателя

### Присваивание пустого значения

```
int *x1 = NULL;  
int *x2 = 0;
```

В первой строке используется константа **NULL**, определенная как указатель, равный нулю.

**Рекомендуется** использовать просто цифру 0, так как это значение типа *int* будет правильно преобразовано стандартными способами в соответствии с контекстом. А так как объекта с нулевым (фиктивным) адресом не существует, пустой указатель обычно используют для контроля, ссылается указатель на конкретный объект или нет

### Присваивание указателю адреса выделенного участка динамической памяти

с помощью операции **new** :

```
int *n = new int;  
int *m = new int (10);
```

с помощью функции **malloc**:

```
int *p = (int*)malloc(sizeof(int));
```

### Присваивание без явного приведения

ТИПОВ

указателям типа **void\***

если тип указателей справа и слева от операции присваивания один и тот же

# Операции над указателями

С указателями можно выполнять арифметические операции **сложения, инкремента (++)**, **вычитания, декремента (--)** и **операции сравнения**

**Арифметические операции** с указателями автоматически **учитывают размер** типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, последовательно размещенными в памяти, например с массивами

Указатель, таким образом, может использоваться в выражениях вида:  $p \# iv$ ,  $\# p$ ,  $p \##$ ,  $p \# = iv$ , где  $p$  – указатель,  $iv$  – целочисленное выражение,  $\#$  – символ операции '+' или '-'.

**Инкремент** перемещает указатель к следующему элементу массива, **декремент** – к предыдущему

Результатом таких выражений является увеличенное или уменьшенное значение указателя на величину  $iv * \text{sizeof}(*p)$ , т.е. если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа.

# Операции над указателями

Текущее значение указателя всегда ссылается на позицию некоторого объекта в памяти с учетом правил выравнивания для соответствующего типа данных. Таким образом, значение  $p \# iv$  указывает на объект того же типа, расположенный в памяти со смещением на  $iv$  позиций.

При сравнении указателей могут использоваться отношения любого вида («>», «<» и т.д.), но наиболее важными видами проверок являются отношения равенства и неравенства («==», «!=»).

**Отношения порядка** имеют смысл только для указателей на последовательно размещенные объекты (элементы одного массива).

**Разность** двух указателей дает число объектов адресуемого ими типа в соответствующем диапазоне адресов, т.е. в применении к массивам разность указателей, например, на третий и шестой элементы равна 3.

**Уменьшаемый** и **вычитаемый** указатели должны принадлежать **одному массиву**, иначе результат операции не имеет практической ценности и может привести к непредсказуемому результату. То же можно сказать и о суммировании указателей.

# Операции над указателями

Значение указателя можно вывести на экран с помощью функции ***printf***, используя спецификацию ***%p*** (*pointer*), результат выводится в шестнадцатеричном виде.

```
int a = 5, *p, *p1, *p2;  
p = &a;  
p2 = p1 = p;  
++p1;  
p2 += 2;  
printf("a = %d , p = %d , p = %p , p1 = %p , p2 = %p .\n", a, *p, p, p1, p2);
```

Приме  
р

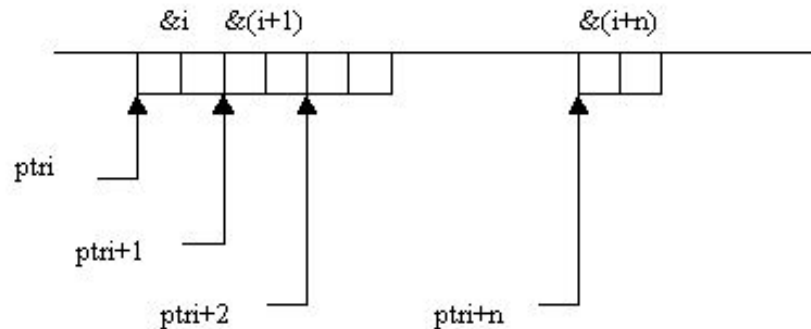
**Результат может быть следующим:**

*a = 5 , \*p = 5 , p = FFF4 , p1 = FFF6, p2 = FFF8 .*

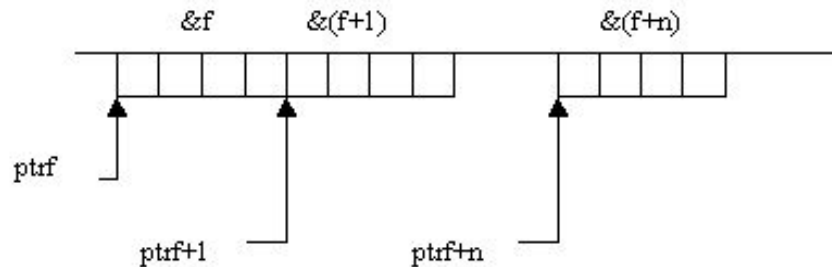
# Операции над указателями

Графически это выглядит следующим образом (в 16-разрядном процессоре на тип *int* отводится 2 байта):

```
int i,*ptri;  
ptri=&i;  
ptri++;
```



```
float f,*ptrf;  
ptrf=&f;  
ptrf++;
```



```
int m[10],*ptrm;  
ptrm=&m[0];  
ptrm++;
```

