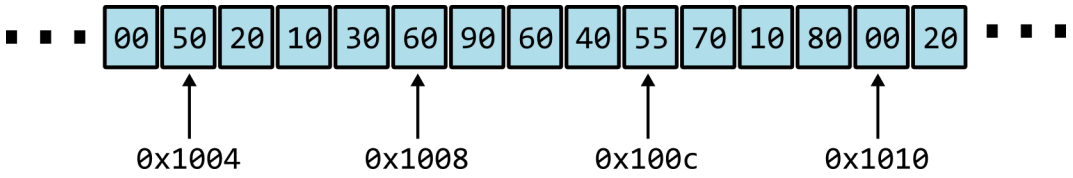
Семинар #4: Часть 1: Указатели.

Память и адреса

Примечание: Оказывается, что при работе с памятью и адресами удобнее использовать шестнадцатеричную систему счисления, поэтому в этом семинаре мы будем использовать её.

Память можно схематично представить как очень большой массив, состоящий из ячеек размером в 1 байт. Каждая ячейка памяти занумерована. По номеру ячейки можно получить доступ к этой ячейке. Например, можно по номеру ячейки прочитать данные из ячейки или записать туда данные. Номер ячейки также называется адресом этой ячейки.

Рассмотрим, например, кусок памяти, в которой хранятся какие-то данные:



На участке памяти, изображённом на картинке:

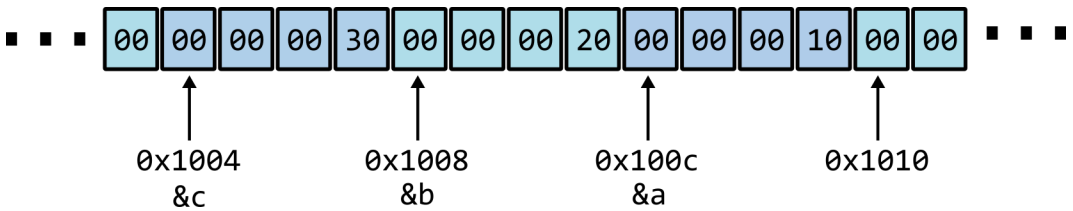
- Ячейка с адресом 0x1004 хранит однобайтовое число 50
- Ячейка с адресом 0x1008 хранит однобайтовое число 60
- Ячейки с адресами от 0x1008 до 0x100c хранят четыре однобайтовые числа 60 90 60 40
- С другой стороны, в последовательности ячеек от 0x1008 до 0x100c можно хранить одно целое число размером в 4 байта (например `int`). Или в той же последовательности ячеек от 0x1008 до 0x100c можно хранить число типа `float`

Адреса переменных

На большинстве систем переменные типа `int` занимают 4 байта. Соответственно, если вы создадите 3 переменные типа `int` вот так:

```
int a = 0x10;   // язык C поддерживает шестнадцатеричную систему счисления
int b = 0x20;   // нужно просто написать 0x перед числом
int c = 0x30;
```

то в памяти это может выглядеть примерно вот так:



Адрес переменной - это адрес первого байта того участка памяти, который занимает данная переменная. Чтобы получить адрес переменной в языке C нужно написать перед именем переменной символ `&`. Для печати адреса с помощью `printf` используется спецификатор `%p`. В этом случае адрес напечатается в шестнадцатеричной системе счисления.

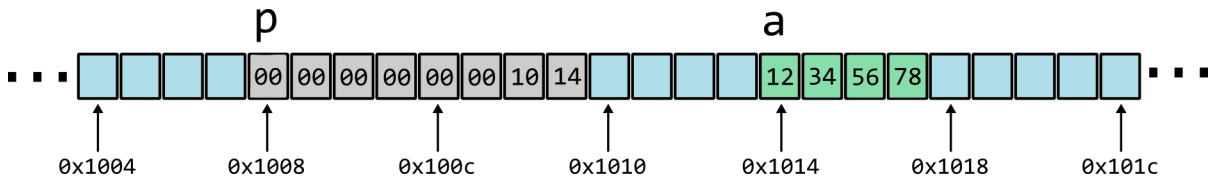
```
int a = 10;
printf("%p\n", &a);
```

Указатели

Для хранения адресов в языке C введены специальные переменные, которые называются указателями. Тип переменной-указателя определяется как тип той переменной, чей адрес он хранит, с добавлением звёздочки. Например, указатель, который будет хранить адрес переменных типа `int`, должен иметь тип `int*`. Размер указателя обычно равен размеру машинного слова, то есть на 64-битных процессорах размер указателя будет равен 8 байтам. Предположим, что определена переменная `a` и указатель `p`, хранящий адрес этой переменной:

```
int a = 0x12345678;
int* p = &a;
```

В памяти эта ситуация может выглядеть так:



Примечания:

- В данном примере для простоты выбраны очень маленькие адреса. В действительности же адрес скорей всего будет очень большим числом.
- Адреса переменных задаются в момент запуска программы и будут разными при разных запусках.
- Указатель тоже является переменной и хранится в памяти.
- На 64-х битных системах указатель обычно занимает 8 байт памяти. Размер указателя не зависит от размера объекта на который он указывает.
- Указатель хранит номер одной из ячеек памяти (в данном случае – первый байт `a`).
- Если указатель хранит адрес какого-то объекта, то также говорят, что он *указывает* на этот объект.

Операция разыменования

Разыменование – это получение самой переменной по указателю на неё. Чтобы разыменовать указатель нужно перед ним поставить звёздочку. То есть, если `p` это указатель, хранящий адрес `a`, то `*p` означает следующее:

Пройди по адресу, хранящемуся в ***p****. Возьми соответствующее количество байт, начиная с этого адреса (в данном случае 4, так как* ***p*** *указывает на* ***int****, а размер* ***int*** *равен 4). Воспринимай эти байты как переменную соответствующего типа (в данном случа* ***int****).*

В примере ниже мы создаём указатель `p`, хранящий адрес переменной `a`. Затем, мы используем `p`, чтобы изменить значение переменной `a`:

```
#include <stdio.h>
int main()
{
    int a = 10;
    int* p = &a;
    *p = 20;
    printf("%i\n", a); // Напечатает 20
}
```

Не следует путать звёздочку, используемую в операторе разыменования со звёздочкой, используемой при объявлении указателя. Это разные звёздочки. Просто так сложилось, что в двух этих случаях используется один и тот же символ.

```
int* p = &a;    // Тут звёздочка используется при объявлении указателя.
                // Она входит в название типа. Тип называется int*

*p = 20;        // Тут звёздочка - это оператор разыменования.
```

Примеры простых программ, использующих указатели

1. Пусть есть переменная `a` типа `float` и указатель `p`, хранящий её адрес. Используем переменную `p`, чтобы возвести `a` в квадрат:

```c
#include <stdio.h>
int main()
{
    float a = 1.5;
    float* p = &a;
    *p *= *p;
    printf("%f\n", a); // Напечатает 2.25
}
```

2. Пусть есть две переменные `a` и `b` и указатель `p`. Используем `p`, чтобы прибавить 1 к обоим переменным:

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    int* p = &a;
    *p += 1;
    p = &b;
    *p += 1;
    printf("%i %i\n", a, b); // Напечатает 11 21
}
```

3. Пусть есть переменная `a` и два указателя `p` и `q`, хранящие адрес переменной `a`. Используем `p` и `q`, чтобы увеличить `a` на 1.

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int* p = &a;
    int* q = &a;
    *p += 1;
    *q += 1;
    printf("%i\n", a); // Напечатает 12
}
```

4. Пусть есть две переменные `a` и `b` и указатель `p`, который хранит адрес `a`. Пусть также есть указателт `q`, который хранит адрес `p`. Используем `q`, чтобы изменить `p` так, чтобы он начал хранить адрес переменной `b`:

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    int* p = &a;
    printf("%i\n", *p); // Напечатает 10

    int** q = &p;
    *q = &b;
    printf("%i\n", *p); // Напечатает 20
}
```

Операции над указателями и арифметика указателей

С указателями можно производить следующие операции:

- Разыменование `*p`
- Проверка указателей на равенство/неравенство: `p == q` и `p != q`

Если указатель указывает на один из элементов массива, то к нему также применимы следующие операции:

- Инкремент `p++`
  В этом случае указатель не увеличивается на `1`, как было можно подумать. Он увеличивается на размер типа, на который он указывает. Таким образом указатель станет указывать на следующий элемент массива.
- Декремент `p--`
  Уменьшается на размер типа, на который он указывает. После этого указатель станет указывать на предыдущий элемент массива.
- Прибавить или отнять число `p + k` и `p - k`
  В этом случае указатель не увеличивается на `k`, как было можно подумать. Он увеличивается на `k * sizeof(*p)`. Если `p` указывает на `i`-ый элемент массива, то `p + k` будет указывать на `(i + k)`-ый элемент.
- Вычитание 2-х указателей `p - q`
  Вернётся разница между указателями делённая на размер типа указателя. При этом важно, чтобы оба указателя указывали на элементы одного и того же массива, иначе эта операция приведёт к ошибке.
- Сравнение 2-х указателей `p > q` и т. д.
  Оба указателя должны указывать на элементы одного массива. Указатель больше, если он указывает на элемент с большим индексом.
- Квадратные скобки `p[i]`
  Также как и к массивам, к указателям можно применять квадратные скобочки. Квадратные скобочки эквивалентны прибавлению числа и разыменованию по следующей формуле: `p[i]` = `*(p+i)`

Используя операторы, приведённые выше важно не выйти за пределы массива. Иначе произойдет ошибка.

Применение арифметики указателей для доступа к элементам массива

```c
#include <stdio.h>
int main()
{
    int a[6] = {10, 20, 30, 40, 50, 60};

    int* p = &a[0];
    printf("%i\n", *p);       // Напечатает 10
    p++;
    printf("%i\n", *p);       // Напечатает 20
    printf("%i\n", *(p + 2)); // Напечатает 40
    printf("%i\n", p[2]);     // Напечатает 40
}
```

Обход массива с помощью указателя

```c
#include <stdio.h>
int main()
{
    int a[6] = {10, 20, 30, 40, 50, 60};
    for (int* p = &a[0]; p <= &a[5]; ++p)
        printf("%i\n", *p);
}
```

Схематическое изображение указателей в памяти

Так как постоянно рисовать переменные в памяти слишком громоздко и затруднительно, будем изображать их схематически. Указатели будем изображать серым прямоугольником, а другие переменные – прямоугольниками другого цвета. Стрелочкой будем указывать на переменную, адрес которой хранит указатель. Размеры прямоугольников не соответствуют размерам переменных.

Например, переменные из следующей программы:

```c
#include <stdio.h>
int main()
{
    int a = 123;
    int* p = &a;
    printf("%i\n", *p); // Напечатает 123
}
```

Будем изображать вот так:

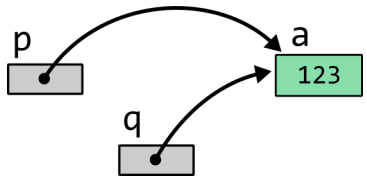
Приведём ещё примеры кода и его соответствующего схематического изображения:

- **Символ и указатель на символ**

```c
char c = 'A';
char* p = &c;
printf("%c\n", *p); // Напечатает A
```

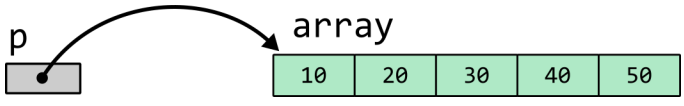
- **Два указателя, которые указывают на одну переменную типа `int`**

```c
int a = 123;
int* p = &a;
int* q = &a;
printf("%i\n", *p); // Напечатает 123
printf("%i\n", *q); // Напечатает 123
```



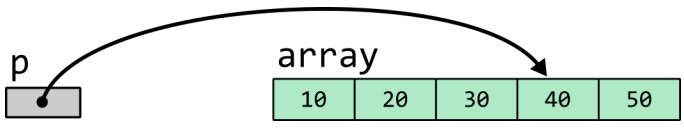
- **Указатель типа `int*`, указывает на первый элемент массива `int`-ов под названием `array`**

```c
int array[5] = {10, 20, 30, 40, 50};
int* p = &array[0];
printf("%i\n", *p); // Напечатает 10
```



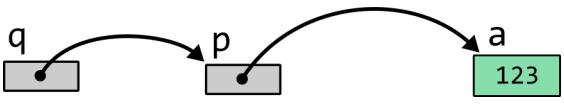
- **Указатель типа `int*`, указывает на четвёртый элемент массива `int`-ов под названием `array`**

```c
int array[5] = {10, 20, 30, 40, 50};
int* p = &array[3];
printf("%i\n", *p); // Напечатает 40
```



- **Указатель типа `int*`, указывает на четвёртый элемент массива `int`-ов под названием `array`**

```c
int a = 123;
int* p = &a;
int** q = &p;
printf("%i\n", **q); // Напечатает 123
```



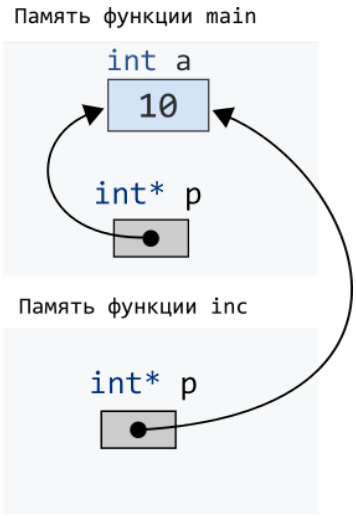
Передача в функцию по указателю

Указатели, как и другие переменные, можно передавать в функции. Например, функция `print_address` из примера ниже принимает на вход указатель на `int` и печатает значение этого указателя на экран:

```c
#include <stdio.h>

void print_address(int* p)
{
    printf("%p\n", p);
}

int main()
{
    int a = 10;
    print_address(&a);
}
```

Как известно, при передаче аргументов в функцию их значения копируются (за исключением массивов). Указатели в этом плане не отличаются от обычных переменных — они тоже копируются. Но что же означает копия указателя? Если указатель хранит адрес некоторой переменной, то и его копия будет хранить тот же адрес. Поэтому, передавая в функцию не саму переменную, а указатель на эту переменную, мы можем менять саму переменную внутри, используя указатель.

```c
#include <stdio.h>

void inc(int* p)
{
    // p хранит адрес переменной a
    *p += 1;
}

int main()
{
    int a = 10;
    int* p = &a;
    inc(&a);
    printf("%i\n", a); // Напечатает 11
}
```



Примеры передачи указателей в функции

1. Функция, которая принимает адреса двух переменных и возвращает их сумму:

```c
#include <stdio.h>

int add(int* pa, int* pb)
{
    return *pa + *pb;
}

int main()
{
    int a = 10;
    int b = 20;
    printf("%i\n", add(&a, &b)); // Напечатает 30
}
```

2. Функция, которая принимает адреса двух переменных и меняет их значения местами:

```c
#include <stdio.h>

void swap(int* pa, int* pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}

int main()
{
    int a = 10;
    int b = 20;

    printf("%i %i\n", a, b);  // Напечатает 10 20
    swap(&a, &b);
    printf("%i %i\n", a, b);  // Напечатает 20 10
}
```

3. Используя указатели в качестве параметров функции, можно добиться того, что функция будет "как бы возвращать" более одного значения. Например, в примере ниже написана функция, которая принимает два числа и "возвращает" 2 значения: минимум и максимум из этих двух чисел.

```c
#include <stdio.h>

void calculate_maxmin(int a, int b, int* pmin, int* pmax)
{
    if (a > b)
    {
        *pmax = a;
        *pmin = b;
    }
    else
    {
        *pmax = b;
        *pmin = a;
    }
}

int main()
{
    int max, min;
    calculate_maxmin(10, 20, &min, &max);
    printf("min = %i, max = %i\n", min, max);

    calculate_maxmin(90, 70, &min, &max);
    printf("min = %i, max = %i\n", min, max);
}
```

Указатели и массивы. Array to pointer decay.

При передаче в функцию массива, туда на самом деле передаётся указатель на первый элемент этого массива.

```
#include <stdio.h>
void func(int a[5])
{
    printf("%zu\n", sizeof(a)); // Напечатает 8, так как a тут это указатель
}
int main()
{
    int x[5] = {10, 20, 30, 40, 50};
    printf("%zu\n", sizeof(x));   // Напечатает 20, так как x это массив из пяти int - ов
    func(x);
}
```

Получается, что когда мы передаём в функцию массив, туда на самом деле передаётся указатель на первый элемент. Можно сказать, что внутри функции массив как бы превращается в указатель. Такое необычное явление в языке называется *array to pointer decay* (*разложение массива в указатель*). При этом, пользоваться указателем `a` в функции `func` можно почти также как и массивом, так как к указателям тоже применимы квадратные скобочки. Чтобы не путать в дальнейшем тип параметра, будем всегда для функций, принимающих массив, явно прописывать тип указателя. Например, функцию `func` из примера выше будем писать так:

```
void func(int* a)
{
    printf("%zu\n", sizeof(a));
}
```

Большой недостаток автоматической передачи массива через указатель заключается в том, что мы никак не можем узнать размер передаваемого массива:

```
void func(int* a)
{
    // Не можем узнать размер массива
}
int main()
{
    int x[5] = {10, 20, 30, 40, 50};
    printf("%zu\n", sizeof(x) / sizeof(x[0])); // Можем узнать размер массива
    func(x);
}
```

Единственный вариант – это передавать размер массива через параметр функции:

```
void func(int* a, size_t n)
{
    printf("%zu\n", n);
}
```

Передача строк в функцию

Строки – это по сути массивы элементов типа `char`. Но конец строки задаётся специальным нулевым символом, поэтому нам не нужно передавать размер строки в функцию отдельным параметром. Мы можем просто узнать размер, например используя функцию `strlen`.

```
void func(char* a)
{
    printf("%zu\n", strlen(a));
}
```

Возврат указателя из функции. Висячие указатели

Указатели можно также возвращать из функций, но делать нужно с осторожностью, так как в этом случае легко написать некорректную программу. . Рассмотрим, например, следующий пример, в котором из функции `func` возвращается адрес локальной переменной `a`.

```
#include <stdio.h>

int* func()
{
    int a = 10;
    int* p = &a;
    return p;
}

int main()
{
    int* q = func();
    printf("%i\n", *q); // Ощибка (UB)
}
```

Эта программа некорректна. Дело в том, что из функции `func` возвращается адрес локального объекта `a`, но все локальные объекты уничтожаются во время выхода из функции. То есть после того как функция `func` отработала, её локальные переменные (`a` и `p`) уничтожатся. В результате в указателе `q` будет хранится адрес уже несуществующей переменной `a`. Такой указатель называют висячим.

Висячий указатель (англ. dangling pointer) – это указатель, указывающий на несуществующий объект.

Разыменование такого указателя приведёт к тому, что вся ваш программа станет некорректной. Возникнет особая ситуация, называемая неопределённым поведением (англ. undefined behavior или UB). Опасность этой ситуации в том, что код может скомпилироваться без предупреждений и ошибок, но программа может работать не так как надо. В результате такую ошибку будет очень сложно обнаружить и исправить.

Тем не менее, есть ситуации, когда возвращать указатель из функции можно. Главное, чтобы объект, адрес которого мы возвращаем, продолжал существовать после завершения функции.

Возвращаем адрес глобальной переменной:

```
#include <stdio.h>

int x = 10;

int* func()
{
    return &x;
}

int main()
{
    int* q = func();
    printf("%i\n", *q); // 10
}
```

Возвращает адрес переменной из памяти `main`:

```
#include <stdio.h>

int* func(int* p)
{
    *p += 1;
    return p;
}

int main()
{
    int x = 10;
    int* q = func(&x);
    printf("%i\n", *q); // 11
}
```

Константные указатели

! Терминология в этом вопросе может отличаться в разных курсах/книгах.

В случае указателя мы можем потребовать сделать неизменяемым две разные величины: сам указатель и объект, на который он указывает. Указатель, который не может меняться сам, но при этом, используя этот указатель, можно поменять то, на что он указывает, будем называть *постоянным указателем*. Такой указатель можно объявить следующим образом:

```
int* const p = &a;
```

Указатель, который можно менять, но, используя его, нельзя изменить то, на что он указывает, будем называть *константным указателем*. Такой указатель можно объявить следующим образом:

```
const int* p = &a;
```

Постоянный указатель

```
int a = 10;
int b = 20;
int* const p = &a;
p = &b;   // Error
*p += 1;  // OK
```

Константный указатель

```
int a = 10;
int b = 20;
const int* p = &a;
p = &b;   // OK
*p += 1;  // Error
```

Как запомнить какой указатель что делает неизменяемым:

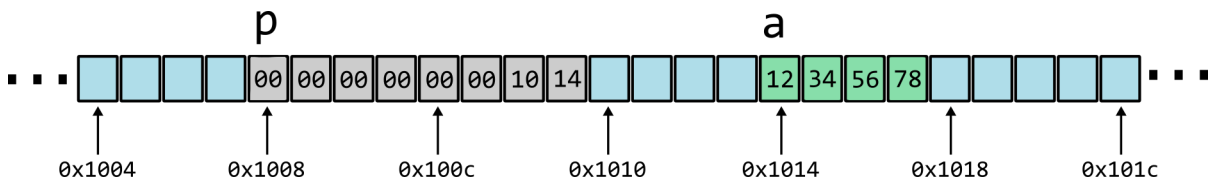
```
int* <@\textcolor{red}{const p}@> = &a;  // p нельзя изменить
<@\textcolor{red}{const int}*@> p = &a;  // int нельзя изменить
```

Указатели разных типов

Как вы могли заметить, тип указателя зависит от типа элемента на который он указывает. Но все указатели, независимо от типа, по сути хранят одно и то же (адрес первого байта объекта).

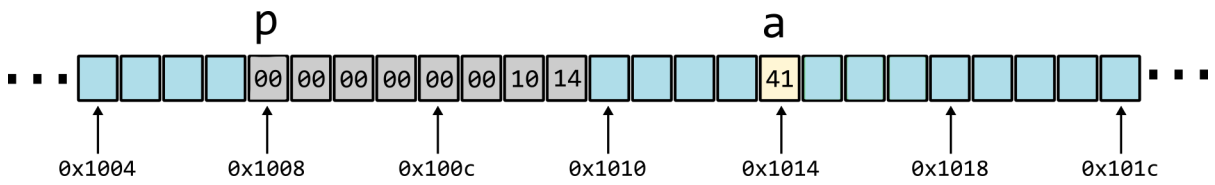
Например, если у нас есть указатель типа `int*`, который указывает на объект типа `int` (размер `int` равен 4 байта), то в памяти это может быть выглядеть примерно так:

```
int a = 0x12345678;
int* p = &a;
```



Если же у нас есть указатель типа `char*`, указывающий на объект типа `char` (размер `char` равен 1 байт), то в памяти это будет выглядеть примерно так:

```
char a = 'A'; // Код символа 'A' равени 0x41
char* p = &a;
```



В чём отличие между указателем типа `int*` и указателем типа `char*`? Разница проявляется в двух случаях:

1. При разыменовывание
   При разыменовывании указатель `int*` берёт 4 байта, начиная с адреса, который он хранит, и воспринимает эти байты как переменную типа `int`. Указатель `char*` берёт 1 байт по адресу, который он хранит, и воспринимает его как переменную типа `char`.

2. При использовании арифметики указателей
   Если прибавить к указателю типа `int*` число `1`, то его значение увеличится на `1 * sizeof(int)`, то есть на 4 байта. Указатель из примера выше после такого будет иметь значение `0x1018`. Если же прибавить `1` к указателю типа `char*` то он увеличится на `1 * sizeof(char)`, то есть на 1 байт. Указатель из примера выше после такого будет иметь значение `0x1015`.

Приведение типов указателей

Указатели можно преобразовывать из одного типа в другой. Это похоже на преобразование типов обычных переменных.

```
int a = 4.1;          // Неявное преобразование из double в int
int b = (int)4.1;     // Явное преобразование из double в int

char* p = &a;         // Неявное преобразование из int* в char* ( не работает в C++ )
char* p = (char*)&a; // Явное преобразование из int* в char*
```

Надо отметить, что язык `C++` строже относится к соблюдению типов, чем язык `C`, и не позволит вам неявно преобразовать указатель одного типа в указатель другого типа.

Указатель `void*`

Особый указатель `void*` не ассоциирован ни с каким типом:

```
int a = 10;
void* p = &a;  // Просто хранит адрес переменной a
```

Такой указазатель нельзя разыменовать и к нему нельзя применять арифметику указателей. Чтобы использовать объект, на который указывает `void*`, нужно сначала привести указатель к необходимому типу:

```
*(int*)p = 20;      // Преобразовали void* в int* и затем разыменовали
printf("%i\n", a);  // Напечатает 20
```

Нулевое значение указателя `NULL`

Иногда нам может потребоваться создать указатель, который бы никуда не указывал. Для таких ситуаций в язык введена специальная константа `NULL`.

```
int* p = NULL;  // Нулевой указатель
```

`NULL` – это такая константа, определённая в стандартной библиотеке с помощью директивы `#define`. Обычно, она определена таким образом, что она имеет тип `void*` и равна нулю.
`NULL` может потребоваться в следующий ситуациях:

- **Проверка ошибок**
  Многие функции возвращают `NULL`, если что-то пошло не так. Поэтому всегда можно проверить: *если указатель равен NULL, значит, ошибка.*

- **Передача в функции**
  Функция может принимать указатель как аргумент, чтобы как-то использовать передаваемый адрес. При передаче в функцию `NULL`, она может это учесть и сделать что-то другое.

- **Инициализация**
  Когда мы только объявили указатель, лучше сразу записать в него `NULL`.

Работать с нулевым указателем нужно осторожно. Разыменование такого указателя приведёт к неопределённому поведению.

Замечания по поводу синтаксиса объявления указателей

Объявление нескольких указателей в одной строке

Допустим мы захотели создать несколько указателей одного типа, например `int*`. Это можно сделать так:

```
int* p;
int* q;
```

Но что если мы хотим сократить эту запись и создать 2 указателя в одной строке, вот так:

```
int* p, q;
```

Кажется, что и `p` и `q` будут иметь тип `int*`, но это не так. На самом деле, в этом случае `p` будет иметь тип `int*`, а `q` будет иметь тип `int`. Чтобы создать два указателя в одной строке, нужно написать так:

```
int* p, * q
```

Или так:

```
int *p, *q;
```

То есть нужно ставить звёздочку перед каждым именем переменной.

Такой странный синтаксис при объявлении указателей существует в языке из-за некоторых решений создателей языка C при его создании. Чтобы не допускать ошибок при объявлении указателей нужно придерживаться всего одного простого правила:

Не объявляйте несколько указателей в одной строке

Если вам нужно создать несколько указателей, то потратьте по одной строке на объявление указателя. Это правило является одним из правил хорошего стиля и относится не только к указателям, но и к переменным других типов: структурам, массивам и даже простым числам. Также это правило хорошего тона верно для большинства других языков программирования (в том числе C++).

Единственное исключение из этого правила - это числовые переменные, которые сильно связаны друг с другом:

```
float x, y, z;          // Допустимо
float distance, speed;  // Недопустимо, так как distance и speed не сильно связаны
```

Разные варианты синтаксиса объявления указателей

Если вы посмотрите код, написанный другими программистами, то увидите, что разные программисты используют разный синтаксис при объявлении указателя. Грубо говоря, разные программисты ставят звёздочку в разных местах при объявлении указателей.
Допустим мы хотим создать указатель типа `int*`. Есть 3 распространённых варианта:

```
int* p;
```

```
int *p;
```

```
int * p;
```

Все эти 3 способа делают одно и то же, а именно создают переменную типа `int*` по имени `p`. Преимущества и недостатки каждого из этих методов:

1. `int* p`
   - Преимущества:
     - Пишем именно то, что происходит. Мы тут пишем, что создаём переменную типа `int*` по имени `p`, это именно то, что происходит.
     - Самый понятный способ, значит, если вы будете использовать этот способ, то и ваш код будет более понятным.
   - Недостатки:
     - Можно ошибиться при объявлении нескольких указателей в одной строке.
     - Не очень удобно объявлять указатели на функции.

2. `int *p`

   - Преимущества:
     - Удобнее объявлять несколько указателей в одной строке.
     - Удобнее объявлять указатели на функции.
   - Недостатки:
     - Пишем не то, что происходит. Кажется что мы тут создаём переменную типа `int`, по имени `*p`, но это не то, что происходит.
     - Ваш код будет менее понятным.

3. `int * p`

   - Преимущества:
     - Нет.
   - Недостатки:
     - Совмещает в себе недостатки двух предыдущих способов.
     - Иногда можно спутать с умножением.

Рекомендации хорошего стиля от создателей языков

- В языке C нет рекомендаций от создателей языка, какой способ использовать. Большинство программистов используют или 1-й или 2-й способ.
- В языке C++ есть рекомендация от создателей языка (называется C++ Core Guidelines). Рекомендуется использовать 1-й способ.

В этом курсе будет использоваться 1-й способ, а также правило хорошего стиля, которое запрещает объявлять несколько указателей в одной строке.