# Лекция 03 — Функции, указатели, массивы и строки

## Оператор множественного выбора `switch`

Конструкция `switch-case` позволяет сравнить некоторое выражение с набором значений, заменяя собой множественный `if-else`:

```
switch(<выражение>) {
    case <const1>:
        <...>;
        break;
    case <const2>:
        <...>;
        break;
    default:
        <...>
        break;
}
```

Некоторые ограничения:

- все константы - различные целые числа;
- каждый блок заканчивается оператором `break`, в противном случае оператор продолжит вычисления для последующих блоков;
- при отсутствии совпадения с какой-либо из констант оператор переходит к вычислению блока `default`, который, впрочем, не является обязательным.

`switch` также предоставляет возможность совмещать условия и определять для них один набор инструкций. Для этого необходимо записать их подряд:

```
switch(<выражение>) {
    case <const1>:
    case <const2>:
    case <const3>:
        <...> // все константы реализуют один код
        break;
}
```

---

# Функции

## Declaration & definition

```
int max(int a, int b); // declaration — объявление

int max(int a, int b) { // definition — определение
    return a > b ? a : b;
}
```

## Функции без вовзращаемого значения

Для функций, не возвращающих никакое значение, в C++ определён тип `void`:

```
void printmessage(){
    std::cout << "I'm a function!\n";
}

int main() {
    printmessage();
}
```

## Функция `main`

`main` — главная функция в программе, всегда возвращая единственное число типа `int` — **код возврата**. В стандарте для этого определены две константы: `EXIT_SUCCESS` и `EXIT_FAILURE.`
`main` умеет работать с аргументами командной строки. Для этого необходимо передать в качестве параметров `argc` (количество аргументов) и `argv` (сами аргументы в виде массива строк):

```
int main(int argc, char** argv) {}
```

При этом, как и любую функцию, `main` можно *перегрузить*[1]:

```
int main(void) {}

int main(int argc, char** argv) {}

int main(int argc, char** argv, other_parameters) {}
```

# "Затемнение" переменных

> ⚠ **Warning**
>
> При совпадении имён переменных более локальные **всегда** "затемняют" более глобальные, т.е. если компилятор встретит в коде программы имя одной из таких переменных, то обратится к наиболее вложенной.

---

# Указатели

> ⓘ **Info**
>
> **Указатель** — переменная, диапазон значений которой состоит из адресов ячеек памяти и специального значения — *нулевого адреса*.
>
> Указатель "указывает" хранящимся внутренним адресом на ячейку памяти, к которой с его помощью можно обратиться.
>
> Значение нулевого адреса используется только для обозначения того, что указатель в данный момент не указывает ни на какую ячейку памяти.

> ⚠ **Warning**
>
> Размер указателя **не зависит** от типа, на который он указывает.

## Операторы разыменования и взятия адреса

Унарный оператор `&` позволяет нам имея переменную получить адрес, по которому она хранится.
Унарный оператор `*` позволяет нам имея адрес переменной получить её значение, т.е. *разыменовать* указатель.

```
int x = 1;
int y = 2;
int z[10];
int* ip; // указатель на int
```

```
ip = &x; // ip теперь указывает на x
y = *ip; // y теперь равен 1
*ip = 0; // x теперь равен 0
ip = &x[0]; // ip теперь указывает на z[0]
```

## Указатели в качестве аргументов функций

Использование указателей позволяет нам получить доступ к переменным, которые находятся вне поля видимости функции. Типичный пример — функция `swap()`:

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Здесь `a` и `b`, которыми мы манипулируем, являются копиями реальных переменных, значения которых мы хотим поменять местами, и существуют лишь пока выполняется функция. Это значит, что исходные значения после вызова `swap()` никак не изменяются. Решение — передать указатели на нужные значения:

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

Здесь мы непосредственно обращаемся к ячейкам памяти, к которым привязаны наши внешние переменные, и изменяем соответствующие значения напрямую.

## `NULL` vs `nullptr`

Мы можем обозначить пустой указатель двумя способами:

1. `nullptr` — литерал из C++, принимающий значение типа `nullptr_t`.
2. `NULL` — макрос из C, обозначающую целочисленную константу 0. Почти всегда он будет неявно приводиться к `nullptr_t` там, где это нужно, но использовать его НЕ НАДО, поскольку при, например, перегрузке функций это может привести к неоднозначному/непредсказуемому поведению компилятора.

## `void*`

Если мы хотим завести указатель на какой-то объект и при этом нам не важен тип этого объекта (или мы не знаем его), мы можем использовать `void*`.
При этом `void*` может быть явно приведен к указателю на любой тип:

```
int i = 238;
int* pi = &i;
void* pv = pi;
int* pj = (int*)pv;
```

> ⚠ **Warning**
>
> Хранимый адрес **не зависит** от типа указателя.

## Указатели на функции

Так как функция так же, как и переменная, явно хранится в памяти, в C++ есть возможность завести указатель на неё. Для этого используется синтаксис вида `<тип> (*<имя функции>)(<параметры>)`:

```
void print() {
    std::cout << "Hello, World!\n";
}

int main() {
    void (*pf)() = print;
    void (*pf2)() = &print;

    pf(2);
}
```

---

# Массивы

## Инициализация

C++ хранит массивы в виде блока *подряд выделенных* ячеек памяти.
Для объявления массива используется синтаксис вида `<тип элемента> <имя массива>[<кол-во элементов>]`. Его можно определить вместе с объявлением, перечислив элементы в фигурных скобках, причем в таком случае не обязательно указывать размерность массива.

Для объявления многомерных массивов достаточно добавить размерность следующего измерения в новых квадратных скобках. Для многомерных массивов так же работают списки инициализации.

```
int arr[10];
int arr2[] = {1, 2, 3, 4, 5};
int arr3[3] = {1, 2, 3};
int arr4[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

## Связь массивов и указателей

Если у нас есть массив `a` и указатель на первый его элемент `pa = &a[0]`, то сдвиг этого указателя позволяет нам получить доступ к произвольному элементу исходного массива. Это возможно благодаря тому, что элементы массива расположены в памяти последовательно.

Дело в том, что при почти любых вычислениях[2] компилятор преобразовывает имя массива в как раз такой указатель ( `a == &a[0]` ), так что мы можем (условно и осторожно) приравнять эти два понятия. Из этого следует, что, например:

- мы можем обращаться к произвольному элементу массива при помощи сдвига указателя `a` и получать его значение разыменованием этого указателя;
- как следствие из предыдущего пункта, семантически `a[i] == *(a + i)`[3];
- мы можем передавать массив в функцию как указатель на первый элемент;
- в случае многомерных массивов обращения `T a[][]`, `T* a[]` и `T** a` эквивалентны между собой (при этом объявлять мы всё ещё можем только первым способом);
- и.т.д.

> ⚠ **Warning**
>
> Имя массива не является указателем в привычном смысле по той единственной причине, что мы *не можем* самостоятельно назначить ему никакой адрес.

## Указатель на массив

Мы также можем завести указатель на весь массив, а не только на первый его элемент:

```
int a[10];
int (*pa)[10] = &a;
```

При этом критически важно понимать, что несмотря на то, что `a` и `&a` указывают на один и тот же адрес, они **не равны** между собой, поскольку имеют разные типы. У `&a` есть ряд важных свойств, которыми не обладает `a`:

- `&a` знает размер массива и указывает не на отдельную ячейку памяти, а на весь блок, занимаемый массивом. По этой причине, например, при инициализации указателя на массив размеры должны совпадать:

  ```
  int a[10];
  int (*pa)[10] = &a; // OK
  int (*pa)[238] = &a; // CE
  int (*pa)[2] = &a; // CE
  ```

- Как следствие из предыдущего пункта, инкремент и декремент `&a` сдвигают значение указателя не на одну ячейку памяти, а на *весь блок*, т.е. на размер массива:

  ```
  int a[10];
  std::cout << std::boolalpha << ((void*)(&a + 1) == (void*)(a + 1)); //
  false
  std::cout << std::boolalpha << ((void*)(&a + 1) == (void*)(a + 10)); //
  true
  ```

---

## Строки

В языке C строка (она же c-style строка) представляет собой обычный массив символов с нуль-терминатором `\0` в конце:

```
char str[] = "Hello world";
```

Символ `\0` сигнализирует о том, что был достигнут конец строки, и всегда дописывается автоматически, что легко проверить с помощью функции `sizeof`:

```
char str[] = "Hello, World!";
std::cout << sizeof(str); // output: 14
```

По определению, строки поддерживают все те же операции и обладают всеми теми же свойствами, что и массивы произвольного типа.

Особая тема, касающаяся c-style строк — это преобразование строковых литералов, принимающих значение `char* const`, в `char*`, т.е. запись вида

```
char* str = "Hello, World!";
```

В C и в C++ до ISO C++11 оно является допустимым, однако начиная с C++11 стандарт перестаёт поддерживать такие касты. Вместо этого предлагается использовать явные преобразования, например, `const_cast`.

---

1. Перегрузка функций в C++ — возможность валидно определить несколько функций с одним названием, но с разными аргументами. ↩
2. В стандарте языка C зафиксировано, что массив поддерживает только два оператора: `sizeof` и `&`. Во всех остальных случаях имя массива приводится к указателю на первый его элемент. ↩
3. На самом деле, это равенство выполняется и на уровне компилятора, т.е. квадратные скобки всегда заменяются сдвигом указателя. Поэтому справа от них вовсе не обязано стоять имя массива, т.е. мы можем, например, написать `(a + 2)[3]` или `2[a]` — это всё будет корректным обращением к некоторому элементу. ↩