Структурное программирование

- Программное обеспечение представляется как набор блоков, выполняемых последовательно.
- Три основных блока: последовательность, ветвление, цикл.
- Запрещается прямой переход между несвязанными блоками.
- Алгоритм программы можно представить в виде блок-схемы.
- Систематизация процесса разработки программного обеспечения, резкий рост количества и качества кода.

Язык C

- Разработан в начале 1970-х годов для нужд разработки операционной системы Unix.
- Является языком, на котором написаны ядра и API большинства современных операционных систем.
- Синтаксис языка C повлиял на развитие большинства современных массовых языков.
- Основная идея — простота реализации компилятора.
- Самый низкий уровень после ассемблера, позволяет обеспечить максимальную производительность и минимальные накладные расходы.

Идентификатор

Идентификатор — это последовательность из символов латинского алфавита и цифр, обозначающих какую либо программную сущность.

Существует только на этапе исходного кода. В процессе компиляции исчезает.

Существуют зарезервированные идентификаторы, используемые для обозначения конструкций языка.

Идентификатор является единственным способом различия программных сущностей для компилятора, одинаковые идентификаторы для разных сущностей не допускаются.

Переменная

Переменная — это область памяти, хранящая данные определенного типа, которой в соответствие поставлен идентификатор (имя переменной).

Имя переменной должно быть уникально в пределах области видимости.

Тип переменной выдается во время ее объявления и не может быть изменен.

Количество создаваемых переменных практически не ограничено.

```
int var; // создает переменную var типа int
         // (хранит целое число)
```

Типы переменных

- **char** — целочисленный тип, 8 бит. Используется для хранения символов в 8-битных кодировках.
- **int** — целочисленный тип. Размер зависит от платформы: обычно 32 бита, иногда 16 бит.
- **float** — число с плавающей точкой, 32 бита (1 бит — знак, 8 бит — порядок, 23 бита — мантисса). IEEE 754.
- **double** — число с плавающей точкой, 64 бита (1 бит — знак, 11 бит — порядок, 52 бита — мантисса). IEEE 754.
- **void** — отсутствие данных. Используется в специальных случаях. Нельзя создать переменную данного типа.

Модификаторы типа

- **short** — указывает, что переменная должна занимать меньше байт, чем обычно.
- **long** — указывает, что переменная должна занимать больше байт чем обычно.
- **unsigned** — указывает, что целочисленная переменная должна быть беззнаковой.
- **const** — указывает, что переменную нельзя изменять.

Примеры переменных

```
char c1;                          // 1 байт, знаковый
                                  // (от -128 до +127)

unsigned char c2;                 // 1 байт, беззнаковый
                                  // (от 0 до 255)

short int i1;                     // 2 байта, знаковый
                                  // (от -32768 до 32767)

unsigned long long int i2; // 8 байт, беззнаковый
                                  // (от 0 до 2^64-1)

const double PI = 3.1416;  // 8 байт с плавающей точкой
                                  // Константа
```

Перечислимый тип

- Вводит новый тип данных.
- Используется для создания именованных целочисленных констант для избежания «Магически чисел» в коде.
- При необходимости преобразуется в целое число и обратно.

Перечислимый тип

```
enum ReturnType  // Ввод нового перечислимого типа
{
    RT_OK,       // 0
    RT_ERROR     // 1
};

ReturnType rt;   // Создание переменной
rt = RT_OK;
// <...>
if (rt == RT_ERROR)
{
    // <...>
}
```

Массивы

- **Массив** — это объединение нескольких однотипных элементов под одним идентификатором.
- При работе с массивами используются **квадратные скобки**.
- В пределах идентификатора переменные отличаются только индексом.
- Нумерация в массиве всегда идет по возрастанию от 0.
- Во время выполнения нет проверки индекса на корректность. При обращении к элементу за пределами массива, операционная система «убьет» приложение.

Массивы

```
char ar[128];     // Массив на 128 элементов
// Индексы в массиве ar: от 0 до 127 включительно

ar[0] = 1;        // Присвоение 1 первому по счету
                  // элементу (с индексом 0).

ar[127] = 32;     // Присвоение 32 последнему по счету
                  // элементу (с индексом 127).

// Отдельные переменные независимы и самостоятельны
int d = ar[4] + ar[12];
```

Указатели

- **Указатель** — это переменная, хранящая адрес другой переменной.
- Адрес — это целое число, уникально определяющее начало области хранения переменной с точностью до байта.
- Тип указателя — это тип данных, адрес которых он хранит.
- При работе с указателями используется символы ***** и **&**.
- Указатель не хранит размер или количество переменных, на которые указывает.
- Указатель на массив и на одиночную переменную не различаются.
- Для работы с данными, адрес которых хранит указатель, необходимо произвести **разыменование** (переход по адресу).

Указатели

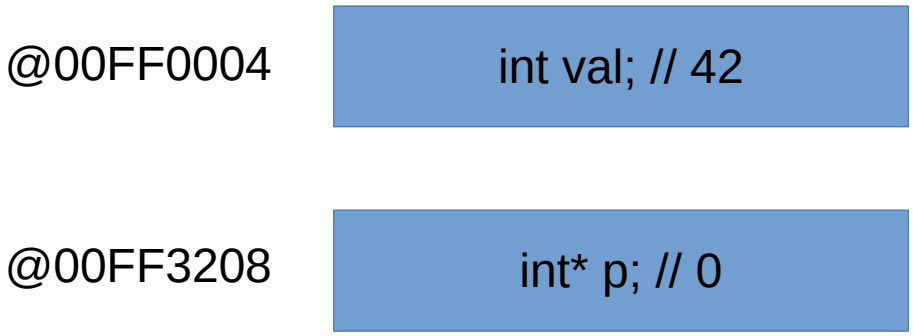
```
int val = 42;
```

@00FF0004

int val; // 42

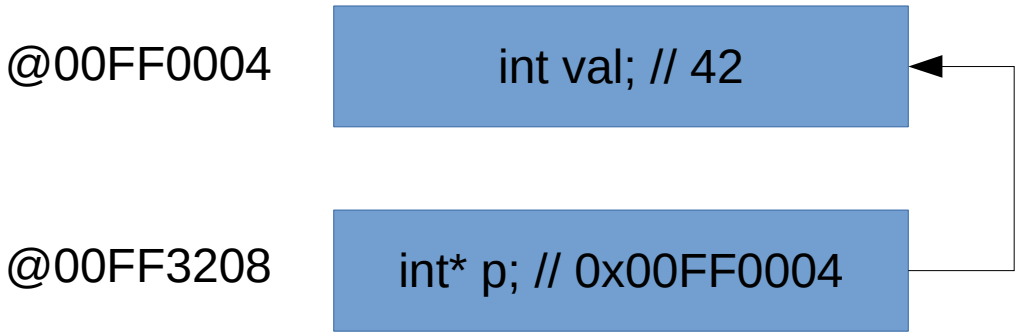
Указатели

```
int val = 42;
int* p = NULL;
```



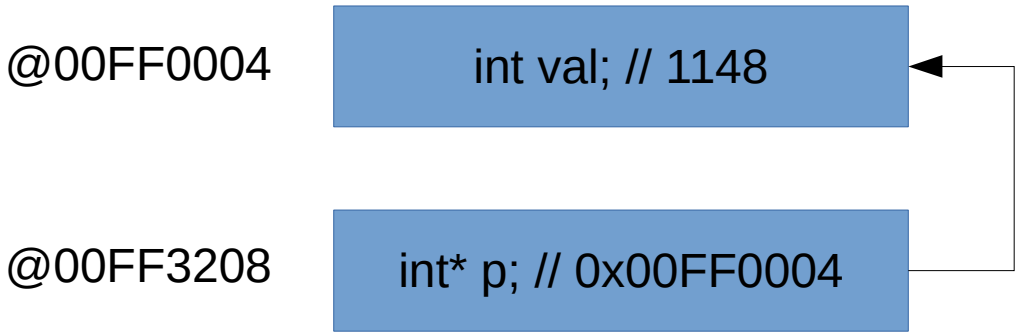
Указатели

```
int val = 42;
int* p = NULL;
p = &val;
```



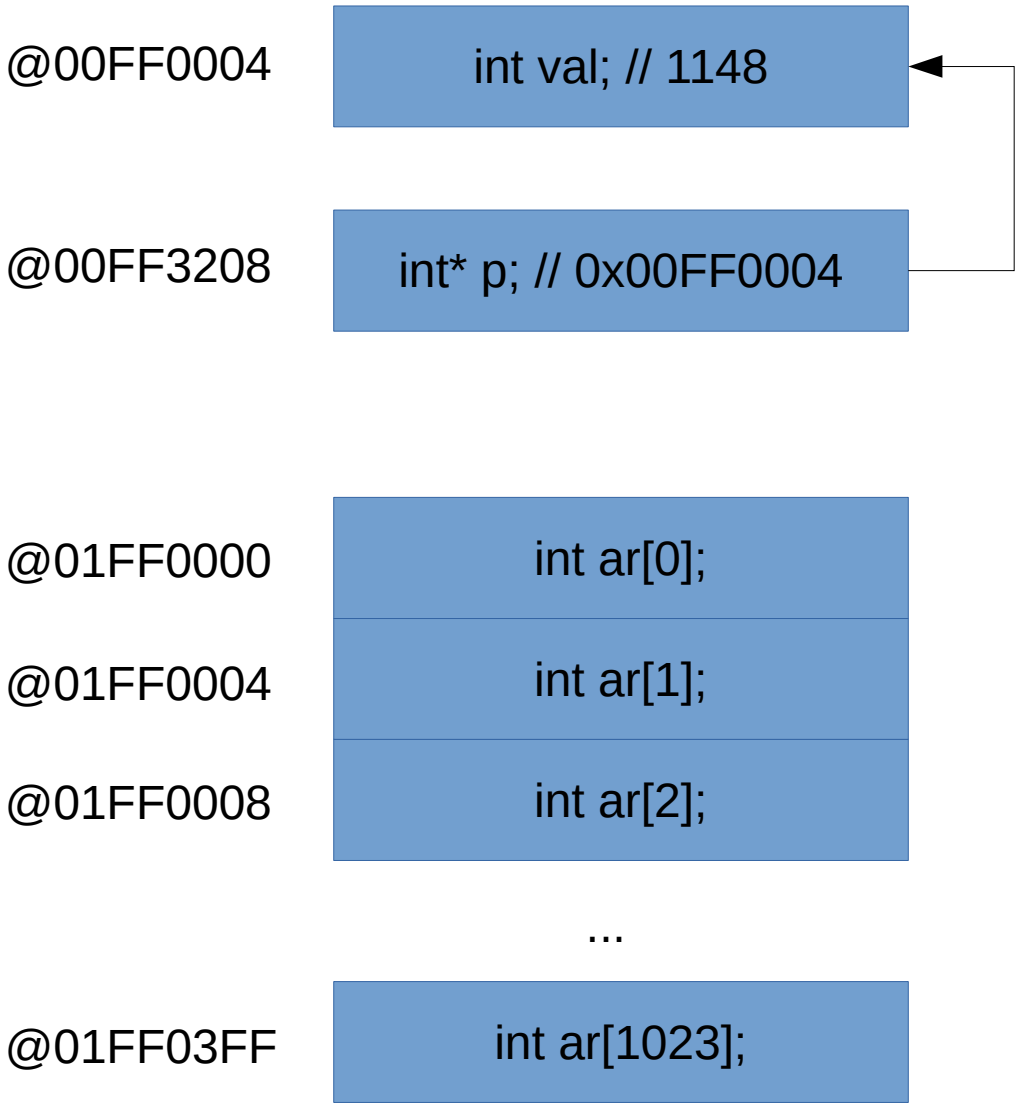
Указатели

```
int val = 42;
int* p = NULL;
p = &val;
*p = 1148;
```



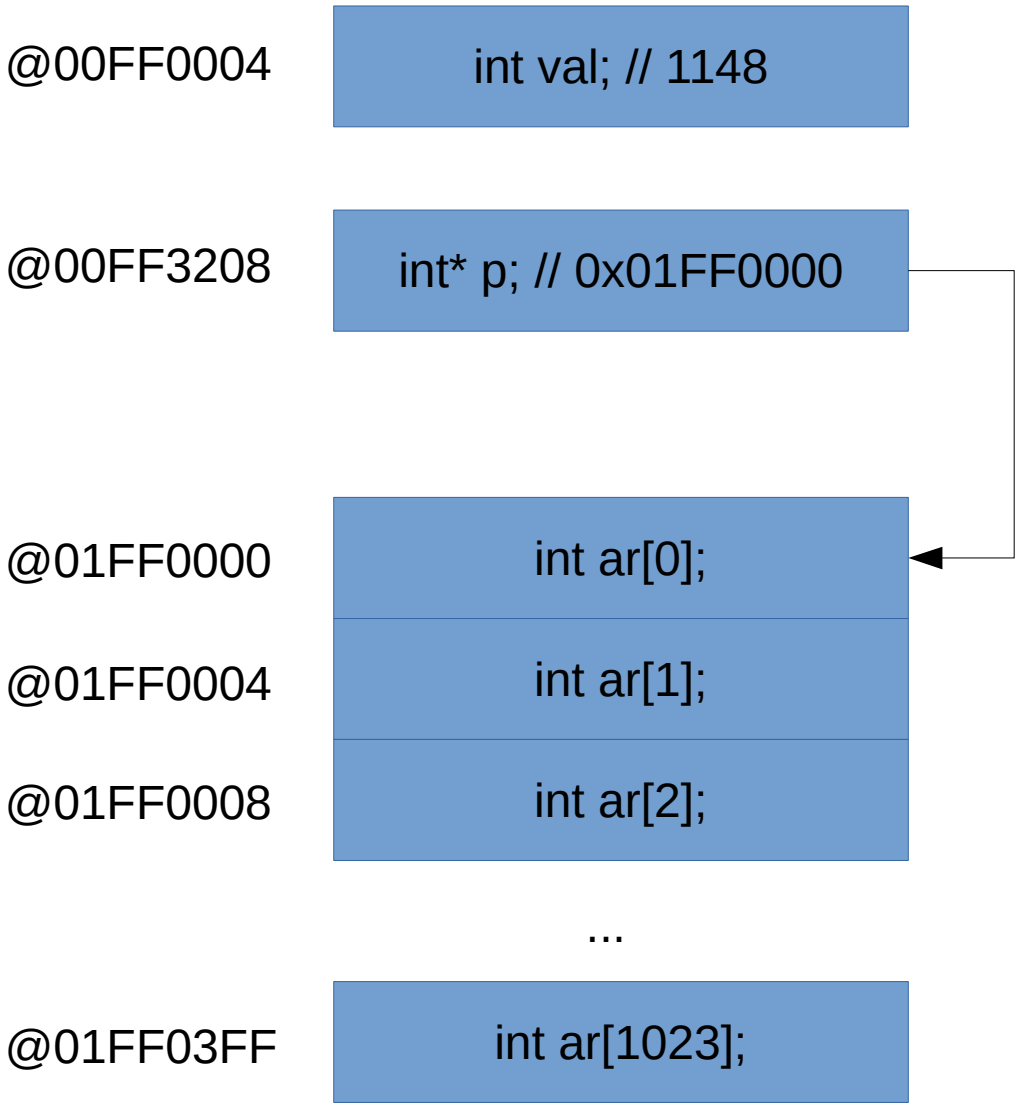
Указатели

```
int val = 42;
int* p = NULL;
p = &val;
*p = 1148;
int ar[1024];
```



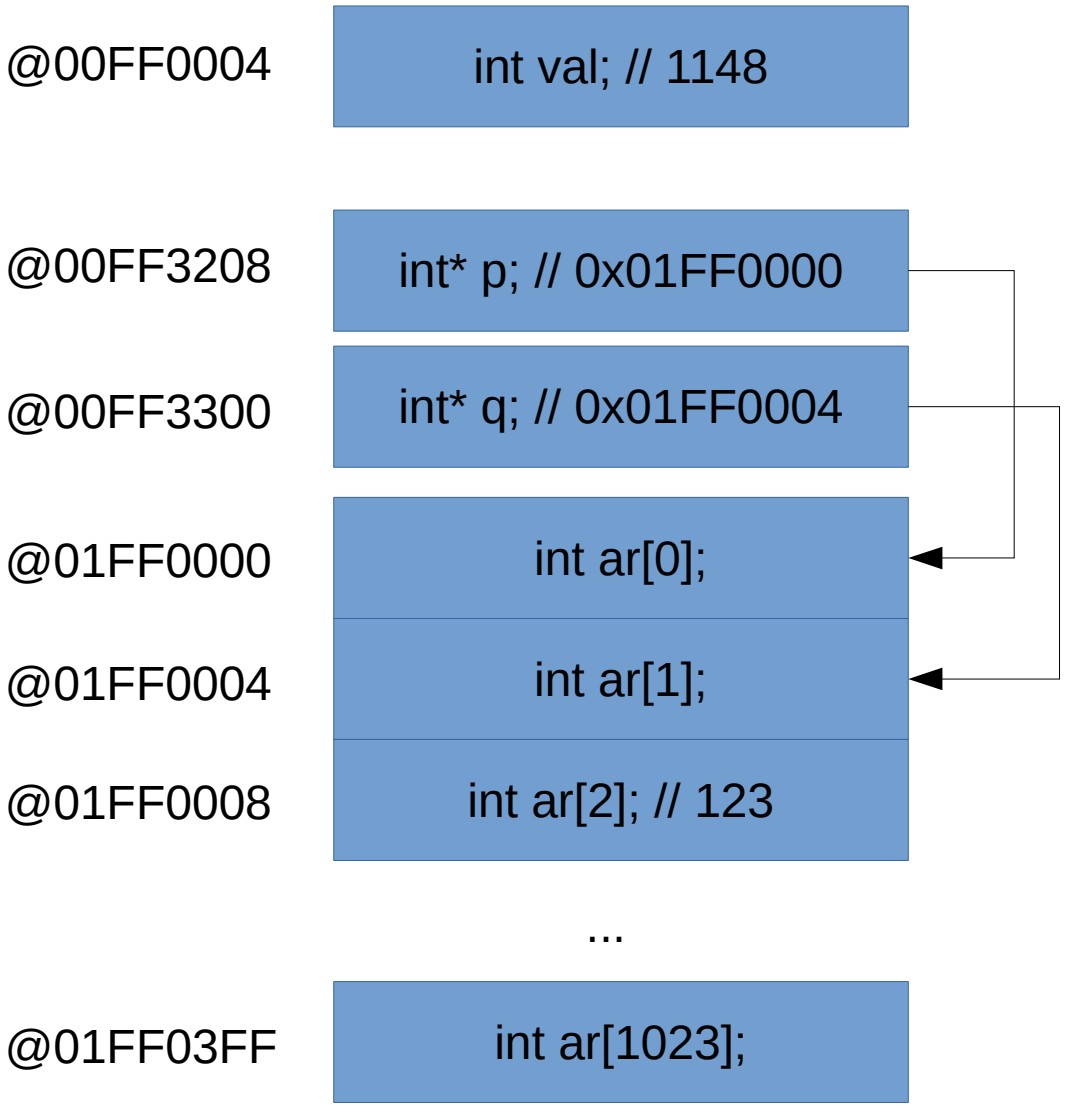
Указатели

```
int val = 42;
int* p = NULL;
p = &val;
*p = 1148;
int ar[1024];
p = ar;
```



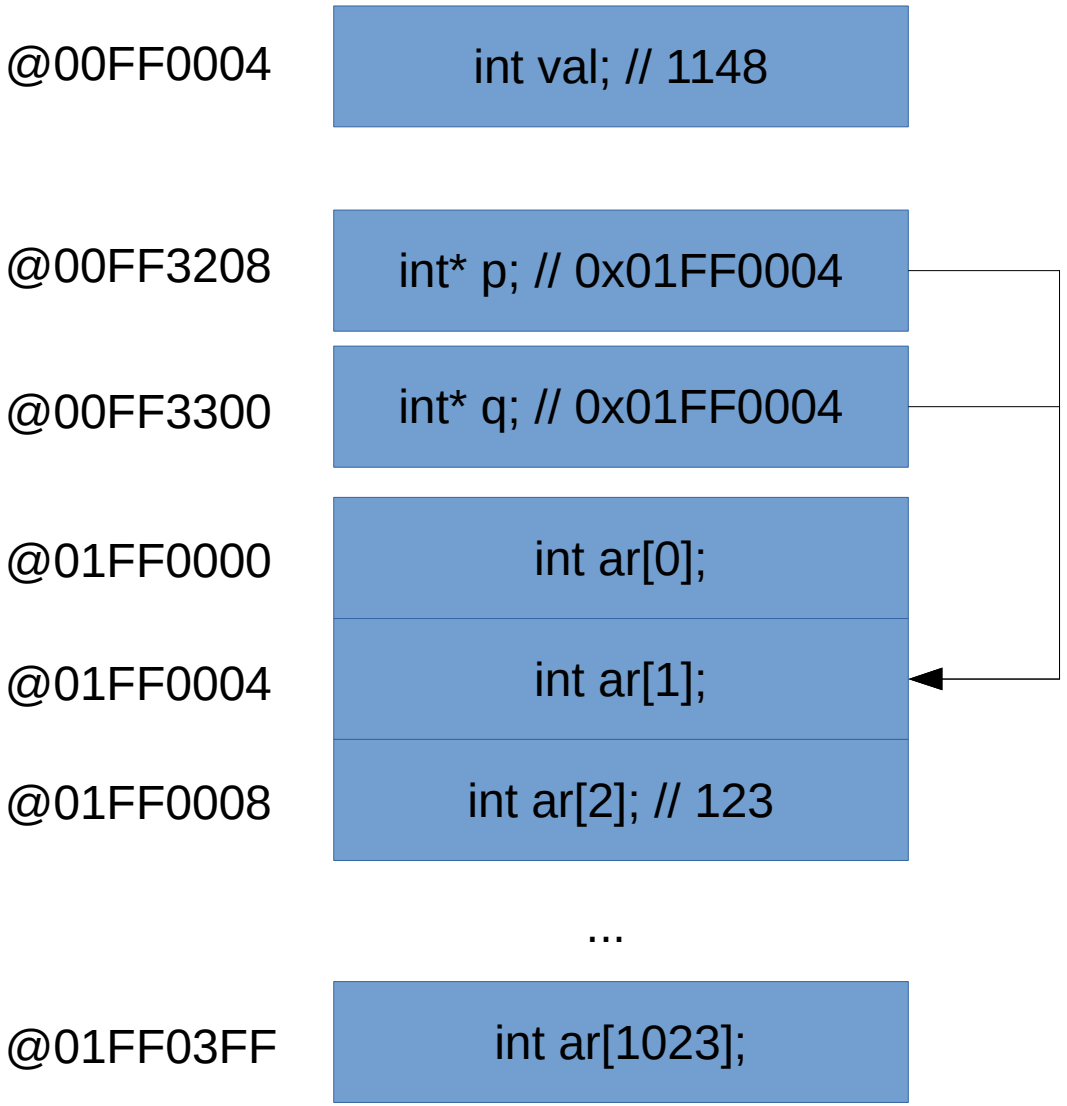
Указатели

```
int val = 42;
int* p = NULL;
p = &val;
*p = 1148;
int ar[1024];
p = ar;
p[2] = 123;
int* q = p + 1;
```



Указатели

```
int val = 42;
int* p = NULL;
p = &val;
*p = 1148;
int ar[1024];
p = ar;
p[2] = 123;
int* q = p + 1;
p = q;
```



Структура

- **Структура** — это тип данных, объединяющий в себе несколько переменных различных типов.
- Структура определяется ключевым словом **struct**.
- Каждая переменная внутри структуры имеет свой идентификатор.
- Доступ к переменным внутри структуры осуществляется через оператор «.» (точка)
- Доступ к переменным, если есть указатель на структуру, осуществляется через оператор ->
- Порядок следования переменных определяет порядок их расположения в памяти => структура используется для описания двоичных форматов данных.

Структура

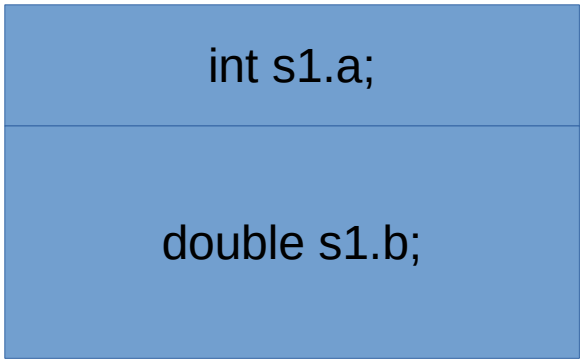
```
struct SS
{
    int a;
    double b;
};
```

Структура

```
struct SS
{
    int a;
    double b;
};

SS s1;
```

@00FF0004

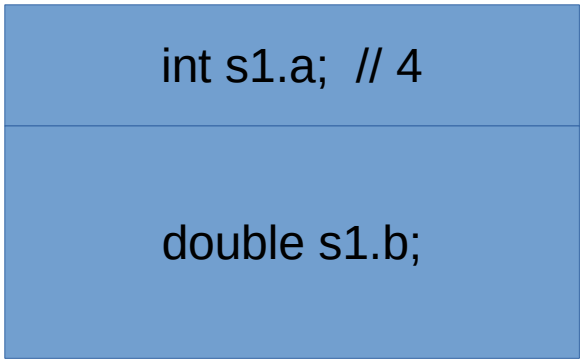


Структура

```
struct SS
{
    int a;
    double b;
};

SS s1;
s1.a = 4;
```

@00FF0004

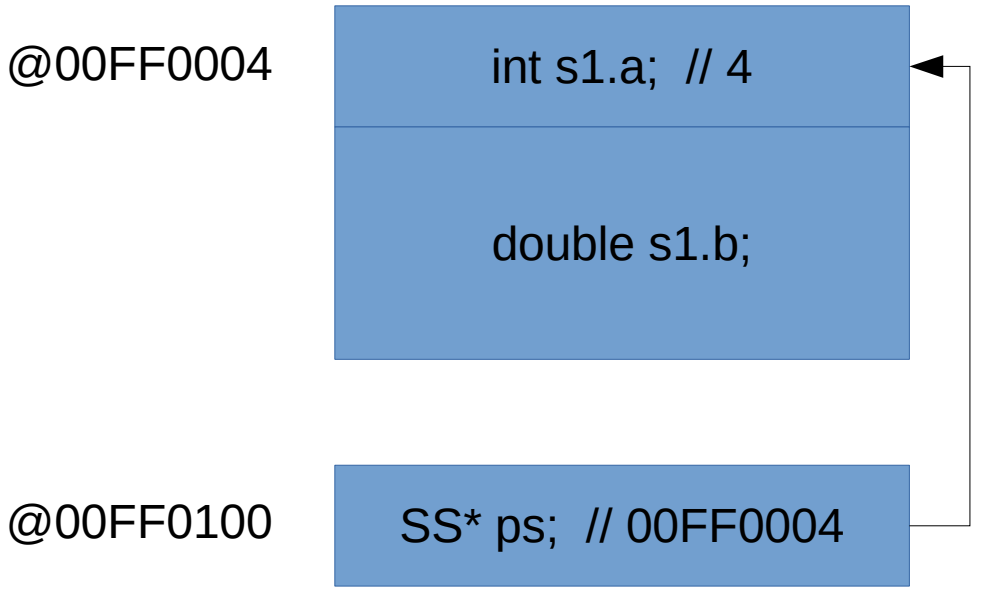


Структура

```
struct SS
{
    int a;
    double b;
};

SS s1;
s1.a = 4;

SS* ps = &s1;
```

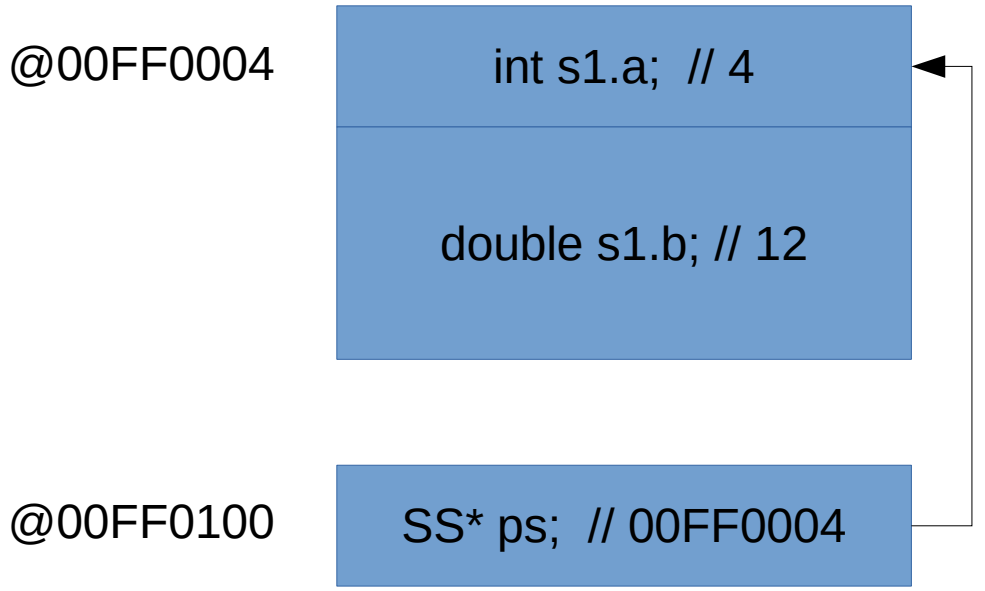


Структура

```
struct SS
{
    int a;
    double b;
};

SS s1;
s1.a = 4;

SS* ps = &s1;
ps->b = 12;
```



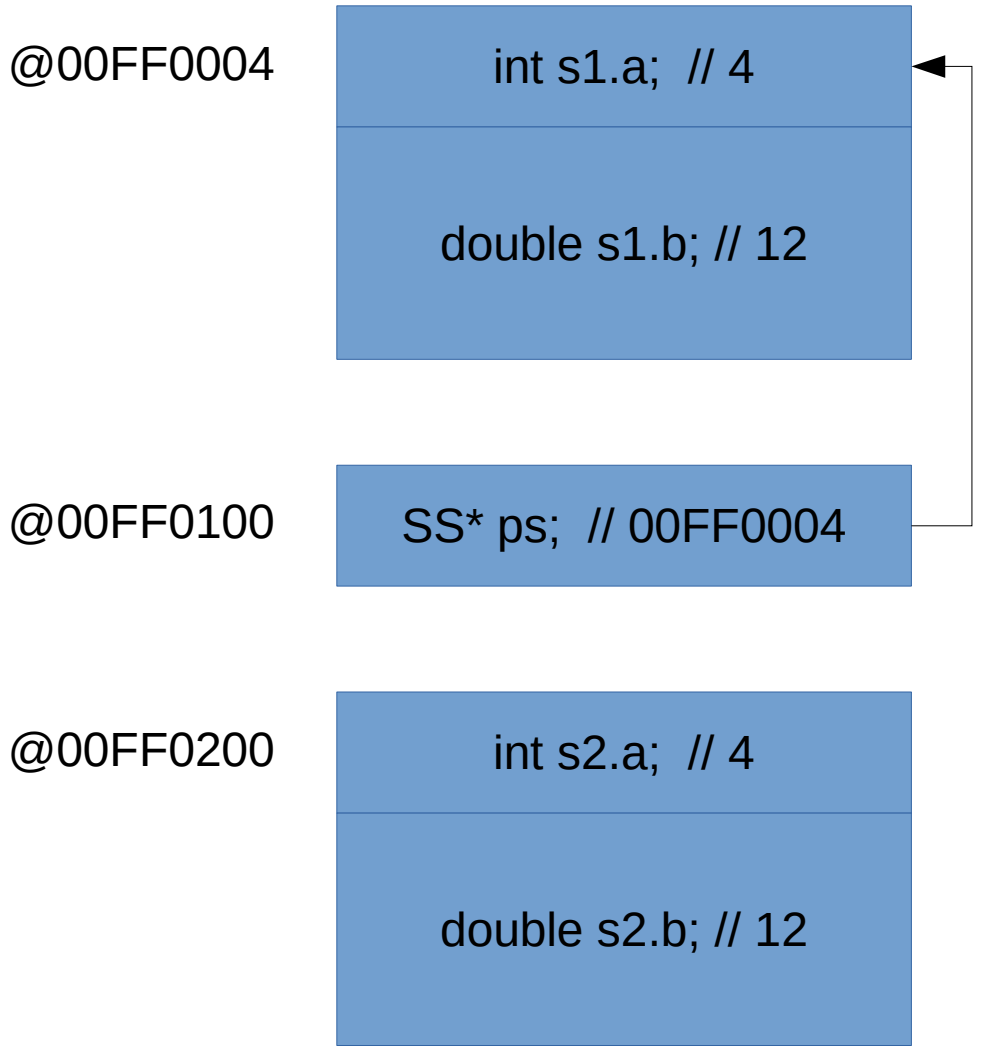
Структура

```
struct SS
{
    int a;
    double b;
};

SS s1;
s1.a = 4;

SS* ps = &s1;
ps->b = 12;

SS s2 = s1;
```



Литерал

- **Литерал** — это способ записи значений переменных в программном коде.
- Значение, заданное литералом, станет частью исполняемого файла после компиляции.
- Литералы бывают **числовые**, **строковые** и **символьные**. Числовые литералы могут записываться в разных системах счисления.

Числовой литерал

Обычная последовательность чисел, является числовым литералом в десятичной системе счисления и соответствует типу int. Например:

```
65342
```

Если литерал начинается с 0x, он определяет число в шестнадцатеричной системе. Если с 0 — в восьмеричной.

```
0171    // 121
0x54BF  // 21695
```

Вещественные числа записываются только в десятичной системе и содержат точку, разделяющую целую и дробную часть. Запись вида <A>e±<B> соответствует чису <A>·$10^{\pm<B>}$

```
.5      // 0.5
1.      // 1.0
6.3e4   // 63000
7.2e-3  // 0.0072
```

Строковый и символьный литерал

Строковый литерал - последовательность любых символов, заключенная в двойные кавычки. Определяет указатель на неизменяемый массив символов, соответствует типу const char*. Например:

```
char* s = "Hello world!";
```

Два идущих подряд строковых литерала склеиваются в один:

```
char* s = "Hello " "world";
```

Символьный литерал — одиночный символ, заключенный в одинарные кавычки. Соответствует типу const char. Может быть преобразован в любой численный тип.

```
char c = 'L';
int charcode = 'L';
```

Escape-последовательности

Символ «\» (обратная косая черта) внутри строкового или символьного литерала определяет специсмвол. Символ «\» вместе со следующим символом преобразуются компилятором в специальный символ.

Последовательность	Результат	Пояснение
\r	<CR>	Перевод каретки
\n	<LF>	Новая строка
\t	<TAB>	Табуляция, «широкий пробел»
\\	\	Вставка символа «\»
\"	"	Вставка символа «"» в строковый литерал.
\'	'	Вставка символа «'» в символьный литерал.

Унарные операторы

- **-** (минус) — обращение знака числа
- **~** (тильда) — обращение всех битов целого числа
- **!** (воскл. знак) — логическое НЕ
- **&** (амперсанд) — получение адреса переменной
- ***** (звездочка) — разыменование указателя
- **sizeof()** — вычисление размера переменной или типа в скобках. В байтах.
- -- (двойной минус) — уменьшить значение переменной
- ++ (двойной плюс) — увеличить значение переменной

Арифметические операторы

- **=** (равно) — присвоение. Всегда происходит справа налево.
- **+**, **-**, ***** — сложение, вычитание и умножение чисел.
- ***/*** (косая черта) — деление чисел. Если первый операнд — целый, то результат также целый (с отбросом дробной части).
- % — остаток от деления целого числа на целое число.

Побитовые операторы

- **&** (амперсанд) — Поразрядное И.

```
0xFFFF & 0x011A      // 0x011A
0x00FF & 0x011A      // 0x001A
```

- **|** (вертикальная черта) — Поразрядное ИЛИ.

```
0x0000 | 0x011A      // 0x011A
0x00FF | 0x011A      // 0x01FF
```

- **^** (крышка) — Поразрядное Исключающее ИЛИ (XOR).

```
0x0001 ^ 0x0101      // 0x0100
0x0100 ^ 0x0101      // 0x0001
```

Побитовые операторы

- **<<** — Сдвиг влево

```
0x0001 << 4      // 0x0010
0x00FF << 8      // 0xFF00
```

- **>>** — Сдвиг вправо

```
0x0100 >> 6      // 0x0004
0x00FF >> 4      // 0x000F
```

Логические операторы

В качестве значения ИСТИНА принимается любое ненулевое число.

В качестве значения ЛОЖЬ принимается любое нулевое число.

- **&&** (двойной амперсанд) — Логическое И

```
0 && 0      // 0
1 && 0      // 0
1 && 1      // 1
```

- **||** (двойная вертикальная черта) — Логическое ИЛИ

```
0 || 0      // 0
0 || 1      // 1
1 || 1      // 1
```

Операторы сравнения

- **==** (двойное равенство) — строгое равенство
- **!=** — не равно
- **<, >** — строгое неравенство
- **<=, >=** — нестрогое неравенство

Для объединения неравенств следует использовать логические операторы, комбинируя отдельные сравнения целиком, при необходимости добавляя скобки:

```
int a = 44, b=12;
a > 10  && a <= 44                  // 1
a < 100 || a > 200                  // 1
(b > 0 && b < 20) || a <100         // 1
```

Приоритет операторов

Каждая операция обладает своим встроенным приоритетом. Например, умножение всегда делается раньше, чем сложение.

Приоритет операций можно изменить, используя скобки, так как они обладают наивысшим приоритетом.

```
10 + 2 * 4          // 18
(10 + 2) * 4        // 48
```

Инструкция

Инструкция — это некое элементарное действие, представляемое как набор операций над переменными и функциями.

Инструкции разделяются символом **;** (точка с запятой).

Синтаксис языка не запрещает расположение нескольких инструкций на одной строке. Однако, для читаемости кода рекомендуется придерживаться правила «одна строка — одна инструкция».

```
int a = 1;
a += 4;
```

Операторный блок

Операторный блок — это единая последовательность инструкций.

Также встречается название «Вычислительный блок».

Для определения блока используются **фигурные скобки**.

Блок выполняется целиком или не выполняется вообще.

Блок определяет **область видимости** для переменных. Любая переменная, объявленная в операторном блоке, существует только в его пределах.

Рекомендуется добавлять отступ к инструкциям внутри блока.

```
{
    int a = 1;
    a += 4;
}
```

Оператор безусловного перехода

Оператор **goto** используется для прямого перехода к определенной ранее метке.

Метка представляет собой идентификатор, после которого записывается двоеточие.

Применение оператора **goto** нарушает принципы структурного программирования, и зачастую просто запрещено.

```
int a = 0;
loop: a += 1;
goto loop;
```

Оператор условного перехода

Оператор **if** используется для вычисления, следует ли выполнять следующий за ним операторный блок.

Условие, записанное в скобках после оператора, вычисляется по правилам логики (0 — ЛОЖЬ, не 0 — ИСТИНА).

Допускается не записывать фигурные скобки. В таком случае в операторный блок входит только первая после оператора **if** инструкция.

После выполнения или невыполнения операторного блока, выполнение программы продолжится далее.

```
int a = 10;
if (a > 0)
{
    a = 0;
}
```

```
int a = 10;
if (a > 0)
    a = 0;
printf("%d", a)
```

Оператор условного перехода

Частая ошибка — вставка точки с запятой после скобок оператора **if**.

В таком случае сразу после оператора **if** расположена пустая инструкция, которая и считается его операторным блоком.

Записанный далее операторный блок не относится к оператору if и будет выполнен в любом случае, независимо от условия.

```
int a = 10;
if (a < 0);
    a = 0;
// Здесь a == 0, хотя условие a < 0 не было выполнено
```

Оператор условного перехода

Вместе с оператором **if** может использоваться оператор **else**. Он ставится сразу после основного операторного блока, и за ним следует альтернативный операторный блок. Альтернативный блок будет выполнен вместо основного, если условие в скобках после **if** не выполнено.

В общем случае ветвь **else** может отсутствовать.

```
int a = 10, b;
if (a > 0)
    a = 0;
else
    b = a;
```

Операторы цикла

Оператор **while** определяет условие, выполнение которого заставляет программу войти в следующий за ним операторный блок, аналогично оператору **if**.

После окончания операторного блока, происходит переход обратно к оператору **while**, и вычисления повторяются.

Как и в **if**, если операторный блок состоит из одной инструкции, фигурные скобки допускается не писать.

Одно выполнение операторного блока носит название **итерация**.

```
int a = 0;
while (a < 10)
{
    a += 1;
}
int b = a;
```

Операторы цикла

Условие в цикле **while** может быть составлено так, чтобы цикл не завершился никогда. В таком случае программа «зависнет», так как будет до бесконечности выполнять данный цикл.

```
int a = 1;
while (a > 0)
{
    a += 1;
}
int b = a; // Эта строка никогда не будет выполнена
```

Операторы цикла

Внутри цикла доступны операторы **break** и **continue**.

break завершает цикл и переводит выполнение программы к первой строке после операторного блока.

continue переводит выполнение сразу к началу следующей итерации.

```
int a = 1;
while (1) // Сюда ведет оператор continue
{
    if (a > 10) continue;
    if (a == 0) break;
    a = a + 3;
}
// Сюда ведет оператор break
```

Операторы цикла

Оператор **while** может записываться после операторного блока, и тогда проверка условия производится после выполнения блока.

Первое выполнение блока происходит всегда.

Перед операторным блоком записывается оператор **do**.

```
int a = 1;
do
{
    a += 2;
} while (a < 10)
```

Операторы цикла

Оператор **for** используется для записи цикла общего назначения.

В скобках **for** через точку с запятой записываются три инструкции:

- **Инициализация**. Инструкция, выполняемая при старте цикла.
- **Условие**. Вычисляется каждый раз при возвращении к **for** и определяет, произойдет ли вход в операторный блок.
- **Инструкция продолжения**. Выполняется каждый раз при возвращении к **for**.

Если операторный блок **for** состоит из одной инструкции, также допускается не записывать фигурные скобки.

Внутри цикла **for** также доступны операторы **break** и **continue**.

Операторы цикла

Оператор **for** используется для короткой записи цикла по элементам массива и прочим множествам, размер которых известен.

Пример иллюстрирует цикл **for** и аналогичный ему цикл **while**.

```
char c[128];
for (int a=0; a<128; ++a)
{
    c[a] = 0;
}
```

```
char c[128];
int a = 0;
while (a<128)
{
    c[a] = 0;
    ++a;
}
```

Функции

Функция представляет собой операторный блок, оформленный таким образом, чтобы его можно было выполнить из другого места программы.

Функции ставится в соответствие идентификатор — имя функции.

Функция может принимать **аргументы**, являющиеся переменными для данного операторного блока. Типы и количество аргументов задаются при объявлении функции.

Функция может вернуть одно значение любого типа. Тип возвращаемого значения задается при объявлении функции.

Для вызова функции необходимо записать ее имя и «вызвать» ее с помощью круглых скобок. Круглые скобки записываются даже при отсутствии аргументов.

Функции

Функция объявляется так же, как переменная, но после ее идентификатора идут круглые скобки, содержащие объявления аргументов через запятую, а дальше следует операторный блок.

Функцию нельзя определить внутри другой функции.

Для возврата значения используется ключевое слово **return**. После вызова **return** программа немедленно выходит из функции, независимо от идущих дальше инструкций.

```
double power(double a, int p)
{
    double res = a;
    for(int a=0; a<p; a++) res *= a;
    return res;
}
```

Функции

Для вызова функции следует написать ее имя и записать пару фигурных скобок, в которых указываются аргументы.

Результат выполнения функции можно сохранить в переменную того же типа, который указан при объявлении функции как возвращаемый.

В программе всегда есть функция **main**. С вызова этой функции начинается выполнение программы операционной системой.

```
int main()
{
    double a=power(2, 10);
    if( a > power(10, 3) )
        a=0;
}
```

Функции

Допускается вызывать только функцию, записанную ранее в том же файле исходных кодов.

Поскольку не всегда есть возможность записать функции в порядке использования, существует **объявление** или **прототип** функции. Это заголовок функции без операторного блока, заканчивающийся точкой с запятой.

Прототип позволяет разместить тело функции в другом месте или даже в другом файле.

```
double power(double a, int p);  // Прототип функции

int main()
{
    double a=power(2, 10);
}
```

Директивы препроцессора

Директивы препроцессора - это команды, обрабатываемые на этапе компиляции.

Директивы записываются маленькими буквами после символа # (решетка).

Директива **#define** определяет правило автозамены. После обработки **#define** будет произведена замена во всем файле.

```
#define M_PI 3.1416 // Замена M_PI на 3.1416

double r = 4;
double s = M_PI * r * r;

// После автозамены:
// double s = 3.1416 * r * r;
```

Директивы препроцессора

Директива **#include** подставляет содержимое файла вместо себя.

Файл после **#include** — это, как правило, заголовочный файл, содержащий прототипы необходимых функций.

Угловые скобки используются после include для включения системного файла.

Двойные кавычки — для включения файла из папки проекта.

```
#include <stdio.h>

int main()
{
  printf("Hello world\n"); // Функция из файла stdio.h
  return 0;
}
```

Сборка проекта

Процесс компиляции исполняемого файла состоит из следующих этапов:

1. Каждый файл «.c» из состава проекта обрабатывается препроцессором.
2. Обработанный файл исходника компилируется в объектный файл, содержащий двоичный код с неразрешенными зависимостями (например, вызов неизвестной данному файлу функции).
3. Полученные объектные файлы, а также известные на момент сборки библиотеки, используются компоновщиком для формирования исполняемого файла. Компоновщик разрешает зависимости путем подстановки адресов вызываемых функций.

Сборка проекта