

# Программирование микроконтроллеров 2

## Основы языка Си

# Обзор инструментов

- В качестве компилятора мы будем использовать тулчейны семейства GCC (GNU Compiler Collection);
- В качестве IDE (Integrated development environment) мы будем использовать Eclipse;
- Для программирования для STM32 мы будем использовать STM32 Cube IDE (на самом деле тот же Eclipse и GCC)



# Программа «Hello world!»

```
*main.c x
1 /*
2  * main.c
3  *
4  * Created on: 9 нояб. 2022 г.
5  * Author: snork
6  */
7
8
9 #include <stdio.h>
10
11
12 int main()
13 {
14     printf("Hello world!\n");
15     return 0;
16 }
17
18
```

# Программа «Hello world!» Комментарии

```
*main.c x
1 /*
2  *.main.c
3  *.
4  *. Created on: 9 нояб. 2022 г.
5  *. Author: snork
6  */
7
8
9 #include <stdio.h>
10
11
12 int main()
13 {
14     printf("Hello world!\n");
15     return 0;
16 }
17
```

Зелёным текстом показан комментарий. Комментарии не видимы для компилятора и предназначены для вас или для других разработчиков.

Все что находится между `/*` и `*/` считается комментарием.

Так же, комментарием считается все от `//` до конца строки

# Программа «Hello world!» Комментарии

Эклипс помечает комментарии зелёным, поэтому легко понять, если что-то пошло не так

```
13  /* Это комментарий */
14
15  /*
16  Это тоже комментарий
17  */
18
19  int /* и вот это комментарий */ x = 0;
20
21  // И даже это комментарий
22
23  а вот это не комментарий
24
--
```

# Программа «Hello world!» Препроцессор

```
*main.c x
1 /*
2  *.main.c
3  *.
4  *. Created on: 9 нояб. 2022 г.
5  *. Author: snork
6  */
7
8
9 #include <stdio.h>
10
11
12 int main()
13 {
14     printf("Hello world!\n");
15     return 0;
16 }
17
```

Следом идет **директива препроцессора** «include».

Директива препроцессора это инструкция для компилятора. Эти инструкции не будут выполняться при работе вашей программы, а будут выполняться при её сборке

Include дословно переводиться как «включить». Эта директива включает в ваш файл содержимое какого-либо другого со всем кодом, который там кто-либо написал.

Здесь мы включаем определения из файла стандартной библиотеки stdio.h.

std от standard, io от input/output.

# Программа «Hello world!» Препроцессор

```
1 // Подключение файла стандартной
2 // библиотеки из системных каталогов
3 #include <limits.h>
4 // Так тоже работает,
5 // но найдется не тот файл!
6 #include "limits.h"
7
8 // Подключение файла платформы
9 // из системных каталогов
10 #include <linux/termios.h>
11 #include <windows.h>
12
13 // Подключение соседнего файла
14 #include "../another-header.h"
15 // И так тоже работает
16 #include <my-header.h>
```

- ▶ /usr/include
- ▶ /usr/lib/gcc/x86\_64-pc-linux-gnu/12.2.0/include
- ▼ /usr/lib/gcc/x86\_64-pc-linux-gnu/12.2.0/include-fixed
  - ▶ limits.h
  - ▶ syslimits.h
- ▶ /usr/local/include

- ▼ src
  - ▼ project
    - ▶ limits.h
    - ▶ main.c
    - ▶ my-header.h
    - ▶ another-header.h

Файлы указанные в "" компилятор ищет сперва рядом, а потом в глобальных папках. Файлы указанные в <> - наоборот

# Программа «Hello world!» Функции

```
*main.c x
1 /*
2  *.main.c
3  *.
4  *. Created on: 9 нояб. 2022 г.
5  *. Author: snork
6  */
7
8
9 #include <stdio.h>
10
11
12 int main()
13 {
14     printf("Hello world!\n");
15     return 0;
16 }
17
```

Дальше начинается функция по имени main.

**Функции**, или **процедуры**, в языке Си являются «действиями», которые программа может выполнять.

Важно не путать это с математическим определением функции. Здесь функция это какое-то простое совершаемое действие.

Например включить свет, проверить почту, посчитать квадратный корень, сложить два числа.

Функция main — особенная функция. Внутри этой функции размещается действие, которое выполняет ваша программа. С точки зрения ОС — ваша программа это тоже какое-то действие.



# Программа «Hello world!» Функции

```
4 void turn_on_the_light()
5 {
6     // ...
7 }
8
9 bool check_mail()
10 {
11     // ...
12 }
13
14 float calc_square_root(float argument)
15 {
16     // ...
17 }
18
19 int sum_number(int number1, int number2)
20 {
21     // ...
22 }
23
```

Функция оформляется так:

тип\_результата имя\_функции([список\_аргументов])  
{  
 [тело функции]  
}

Как и везде в Си регистр имеет значение. **function** и **fUnCtioN** — это разные функции.

Тип результата это тип того, что получается в результате действия.

Имя функции это её уникальный идентификатор. Нельзя иметь две функции с одним и тем же именем в программе (ну, без уловок).

Аргументы функции это параметры действия, которое делается. Например — числа, которые нужно сложить. Или почтовый ящик, который нужно проверить.

# Программа «Hello world!» Функции

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello world!\n");
6     return 0;
7 }
8
```

```
21
22 int main()
23 {
24     printf("Hello world!\n");
25     return 0;
26
27     printf("Вот это напечатано не будет");
28 }
29
```

Функция `main` не имеет аргументов и возвращает целое число (`int` от `integer`).

В теле нашей функции `main` два действия.

Первое действие это вызов функции `printf`. Второе действие это вернуть вызывающему значение 0.

После каждого действия нужно писать точку с запятой. В си это обязательно

Оператор `return` возвращает не только значение, не и управление. После него выполнение функции завершается

# Программа «Hello world!» Функции

Вне тела функции писать код (то есть вызывать другие функции) - нельзя

```
4 |  
5 printf("Привет, мир!\n");  
6  
7 int main()  
8 {  
9     return 0;  
10 }  
11  
12
```

# Типы данных языка Си

## Целочисленные типы – типы, способные хранить целые числа

**char** (1 байт == 8 бит);

signed: [-128, +127], unsigned: [0, +255]

**short int, short, int** ( $\geq 2$  байта == 16 бит)

signed: [-32767, +32767], unsigned: [0, +65535]

**long int, long** ( $\geq 4$  байта == 32 бит)

signed: [-2147483647, +2147483647], unsigned: [0, +4294967295]

**long long int, long long** ( $\geq 8$  байт == 64 бит)

signed: [-9223372036854775807, +9223372036854775807], unsigned: [0, +18446744073709551615]

# Типы данных языка Си

Чтобы не путаться с именами, следует использовать определения из файла `stdint.h` (`#include <stdint.h>`)

`int8_t`: [-128, +127]

`uint8_t`: [0, +255]

`int16_t`: [-32767, +32767]

`uint16_t`: [0, +65535]

`int32_t`: [-2147483647, +2147483647]

`uint32_t`: [0, +4294967295]

`int64_t`: [-9223372036854775807, +9223372036854775807]

`uint64_t`: [0, +18446744073709551615]

# Типы данных языка Си

## Числа с плавающей точкой

### **float**

4 байта (32 бита)

$[-3.4e38, +3.4e38]$

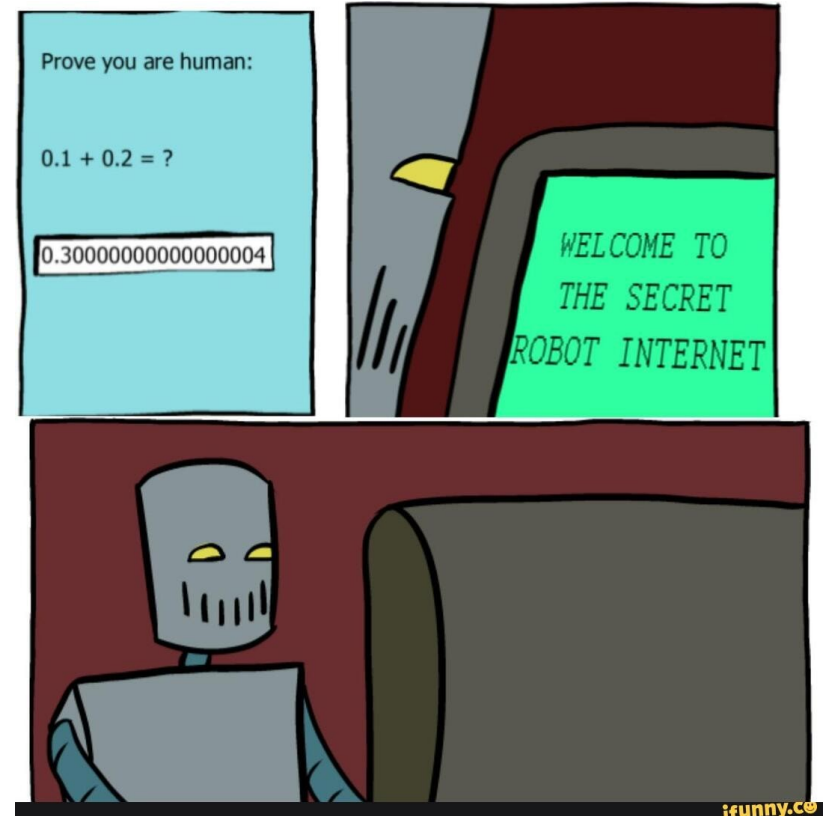
примерно 7 знаков точности

### **double**

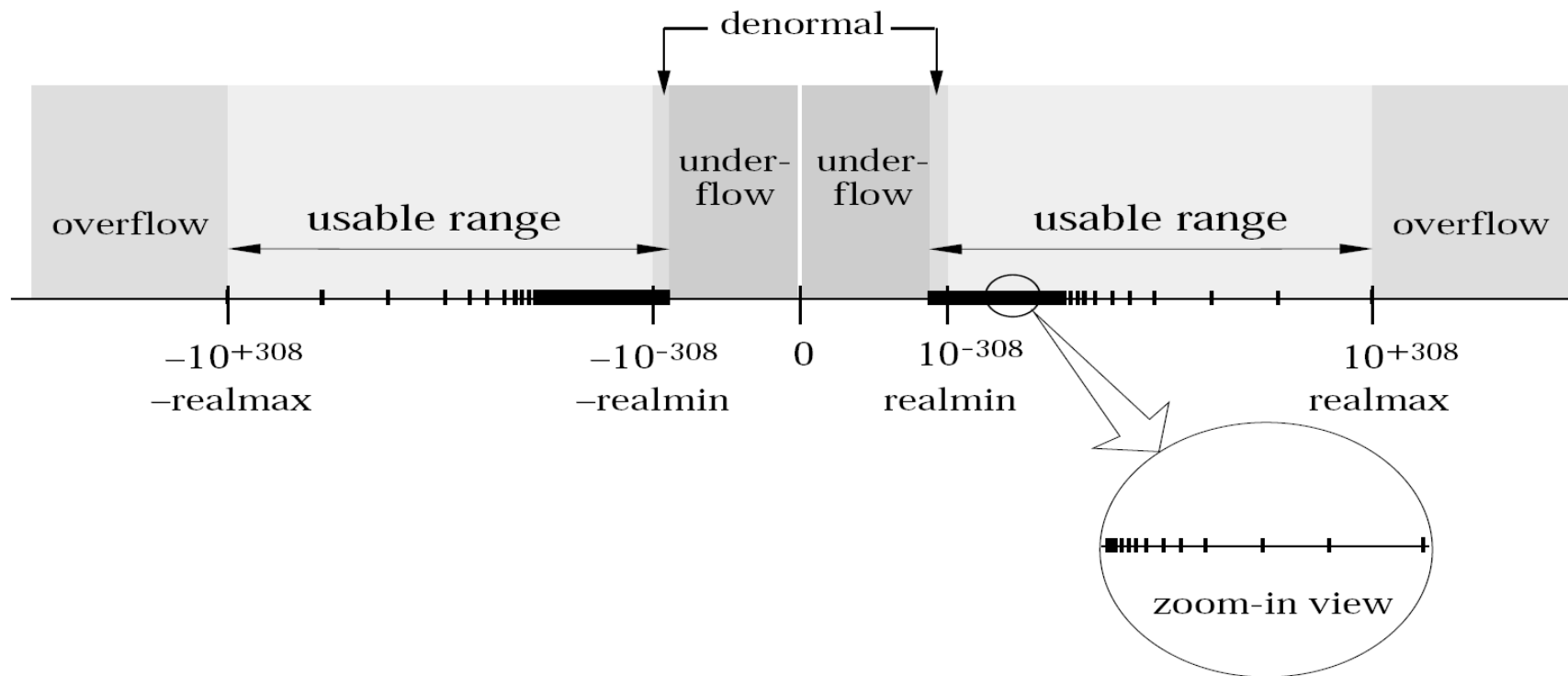
8 байт (64 бита)

$[-1.7e308, +1.7e308]$

Примерно 15 знаков точности.



# Типы данных языка Си



# Типы данных языка Си

VOID — пустота, ничто

```
4 void deploy_parachute()
5 {
6     return; // Здесь нет значения чтобы вернуть,
7     // но управление вернуть мы можем
8
9     // return можно не писать вовсе
10    // функция просто дойдет до конца
11 }
12
13 int function_without_args(void)
14 {
15     // Можно использовать void
16     // чтобы явно показать,
17     // что у функции нет аргументов
18
19     return 0; // Здесь return нужен
20 }
21
22 int function_with_void_arg(void arg0)
23 {
24     // А вот так делать нельзя, это ошибка
25 }
```



# Вызов функции

```
4 void print_number(int number)
5 {
6     printf("Привет, число = %d\n", number);
7 }
8
9 void print_hello()
10 {
11     printf("Привет просто так\n");
12 }
13
14 int main()
15 {
16     print_number(10);
17     print_number(20);
18     print_number(30);
19     print_hello();
20
21     return 0;
22 }
```

Для вызова функции нужно написать её имя и в скобках передать ей аргументы.

Если функция не ждёт аргументов, скобочки все равно нужно написать

# Вызов функции

```
4 void print_number(int number)
5 {
6     printf("Привет, число = %d\n", number);
7 }
8
9 void print_hello(void)
10 {
11     printf("Привет просто так\n");
12 }
13
14 void print_hello_no_void()
15 {
16     printf("У меня явно не указано отсутствие аргументов\n");
17 }
18
19 int main()
20 {
21     // Так делать нельзя
22     print_number();
23     print_number(20, 40, 50);
24     print_hello(123);
25     print_hello_no_void(123, 123, 123);
26
27     return 0;
28 }
```

Нельзя указывать не  
верное количество  
аргументов

# Функция printf

Это функция для вывода текста в терминал. Print – печать. F – formatted. Функция волшебная – она может принимать различное количество аргументов различных типов. Аргументы функции – это различные значения, которые так же нужно выводить на печать.

printf(«формат», аргумент\_1, аргумент\_2, ...);

```
~
4 int main()
5 {
6     printf("целое = %d, дробное = %f\n", 42, 42.42);
7     return 0;
8 }
9
```

# Функция printf

Базовые спецификаторы функции printf

Код    Формат

%c    Символ

%d    Десятичное целое число со знаком

%e    Экспоненциальное представление числа (в виде мантиссы и порядка) (е на нижнем регистре)

%f    Десятичное число с плавающей точкой

%s    Символьная строка

%u    Десятичное целое число без знака (неотрицательное)

%x    Шестнадцатеричное без знака (строчные буквы)

%X    Шестнадцатеричное без знака (прописные буквы)

%p    Выводит указатель

%%    Выводит знак процента

Это лишь базовые спецификаторы. Возможностей форматирования функций printf / scanf намного больше.

# Переменные

```
1 #include <stdio.h>
2
3 // Функция возвращает напряжение на датчике давления
4 int get_pressure_sensor_millivolts(void)
5 {
6     return 10; // Пока что пускай всегда возвращает 10
7 }
8
9 int main()
10 {
11     // Определение переменной
12     int raw_mv;
13     // Присвоение ей значения
14     raw_mv = get_pressure_sensor_millivolts();
15
16     // Определение другой переменной с присвоением ей значения
17     float pressure = (raw_mv - 100) * 1000 + 100;
18
19     // Вывод на экран
20     printf("Давление = %f\n", pressure);
21     return 0;
22 }
```

Переменные – это хранилище данных определённого типа. Если Вашей программе нужно что-то запомнить на время, то это нужно записать в переменную.

Определение переменной делается так:

**тип\_переменной  
имя\_переменной [= начальное  
значение];**

Начальное значение сразу указывать не обязательно, но нужно указать потом

# Операции с переменными

## Присваивание

```
int x;  
x = 10;
```

## Присваивание цепочкой

```
int x, y;  
x = y = 10;
```

## Сложение вычитание, умножение, деление

```
int x;  
x = 10 + 10;  
x = x + 10 + y + SOME_MACRO;  
x += 10; // тоже самое что x = x + 10;  
Аналогично с умножением *, делением /, вычитанием —,
```

Порядок действий как в математике, можно использовать скобочки.

**Нельзя использовать переменную, которой не присвоено значения для чтения из нее!**

# UNDEFINED BEHAVIOR

Допустимое неопределённое поведение варьируется от полного игнорирования ситуации с непредсказуемыми последствиями до демонов, вылетающих из вашего носа.

Джон Вудс (John F. Woods), comp. std. с 1992

Неопределённое поведение — пожалуй, один из самых печально известных назальных демонов. Оно берёт своё начало в языке C. Коротко говоря, такое поведение делает бесполезной **всю программу**, если нарушаются определённые правила языка.

Неопределённое поведение (англ. undefined behaviour) — свойство некоторых языков программирования в определённых ситуациях выдавать результат, зависящий от реализации компилятора. Допускать такую ситуацию в программе считается ошибкой, даже если на некотором компиляторе программа успешно выполняется. Такая программа не будет кроссплатформенной и может приводить к сбоям на другой машине, в другой ОС и даже на других настройках компилятора.

# UNDEFINED BEHAVIOR

```
4 int main()
5 {
6     int variable;
7     printf("Просто вывод %d\n", variable); // <- UB!
8     return 0;
9 }
```

```
../src/project/main.c: В функции «main»:
```

```
../src/project/main.c:12:9: предупреждение: «variable» is used uninitialized [-Wuninitialized]
```

```
12 |         printf("Просто вывод %d\n", variable); // <- UB!
```

```
    |         ^~~~~~
```

```
../src/project/main.c:11:13: замечание: «variable» было объявлено здесь
```



# UNDEFINED BEHAVIOR

```
5 int main()  
6 {  
7     int8_t x = 100;  
8     x = x + 1000000; // UB!  
9  
10    int16_t xx = 30000;  
11    int8_t xxx = xx; // UB!  
12  
13    uint8_t y = 200;  
14    y = y + 1000000; // А вот так можно  
15    return 0;  
16 }
```

make gcc

Building file: ../src/project/main.c

Invoking: GCC C Compiler

gcc -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP .

Finished building: ../src/project/main.c

# Операции с переменными

## Деление интов — целочисленное

```
float y = 11 / 2; // y == 5.0;
```

Для того, чтобы деление было дробным следует использовать явное преобразование к дробному типу

```
float y = (float)11 / 2; // y == 5.5
```

```
float y = 11 / 2.0; // Так тоже можно
```

```
float y = (float)int_var / 2; // К переменным это тоже относится
```

Так же доступна MOD операция

```
Int y = 12 % 10; // y == 2
```

# Операции с переменными

## Виды литералов и системы счисления

```
5
6 int main() {
7
8     {
9         double a = 5.0;
10        float b = 5.0f;
11        int c = 5;
12        long d = 5L;
13    }
14
15    // аналогично
16    {
17        double a = (double)5;
18        float b = (float)5;
19        int c = (int)5;
20        long d = (long)5;
21    }
22
23    return 0;
24 }
```

```
5 int main()
6 {
7     int dec = 255; » // Десятичная система
8     int hex = 0xFF; » // Шестнадцатеричная
9     int oct = 0377; » // Восьмеричная
10    int bin = 0b11111111; » // Двоичная
11
12    int equal = ((dec == hex) && (hex == oct) && (oct == bin));
13    // equal == 1
14    return 0;
15 }
```

# Операции с переменными

## Логические операции

```
Int x = 10;  
Int result = x > 100; // В итоге result = 0;  
result = x < 100; // В итоге result = 1;
```

## Возможны операторы

- > больше,
- < меньше
- >= больше или равно
- <= меньше или равно
- == равно
- != не равно

Нельзя делать цепочки

```
result = 0 <= x <= 100; // так нельзя
```

**При сравнении знаковых и беззнаковых — знаковые приводятся к беззнаковым (это не UB, но неприятно:  $-1 > 2U$ )**

# Операции с переменными

## Логические операции

### && (AND или же операция «и»)

```
int result1 = 10 >= 5; // result1 = 1
int result2 = 10 <= 5; // result2 = 0
int result3 = result1 && result2; // result3 = 0;
```

### || (OR или же операция «или»)

```
result3 = result1 || result2; // result3 = 1;
```

### ! (NOT или же операция «не»)

```
result3 = 1;
result3 = !result3; // result3 = 0;
```

Можно делать цепочки `int result4 = (result3 || result2) && result1;`

Порядок действий: `! > && > ||`

Для более удобной работы с булевой логикой  
Есть файл `<stdbool.h>`

В нем определены значения **true** (==1) и **false** (==0) А так же тип **bool** (==int), который хранит эти значения.

В целом, работа с логикой в Си очень проста  
Все что 0 – считается **false** Все остальное –  
считается **true**

# Операции с переменными

## ~ (Побитовый ! (NOT) )

```
uint8_t x = 243; // в двоичной системе 1111 0011
```

```
bool cmp = (~x == 12); // ~x в двоичной системе == 0000 1100
```

## & и | (Побитовые && и ||)

Аналогично небитовым

```
uint8_t x = 0b11110000;
```

```
uint8_t y = 0b00111100;
```

```
x | y == 0b11111100;
```

```
x & y == 0b00110000;
```

## >> и << (Побитовые сдвиги)

```
uint8_t x = 0b0001;
```

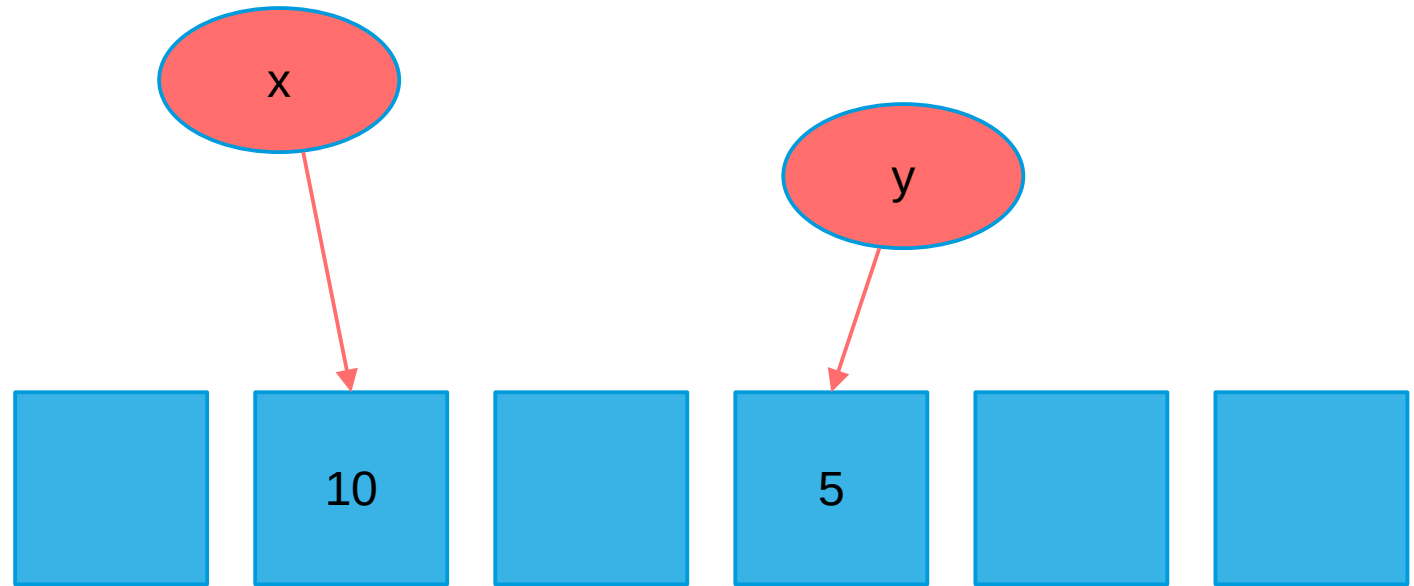
```
x = x << 2 // x == 0b0100;
```

```
X = x >> 4 // x == 0b0000;
```

# Переменные - бирки

`x = 10;`  
`y = 5;`

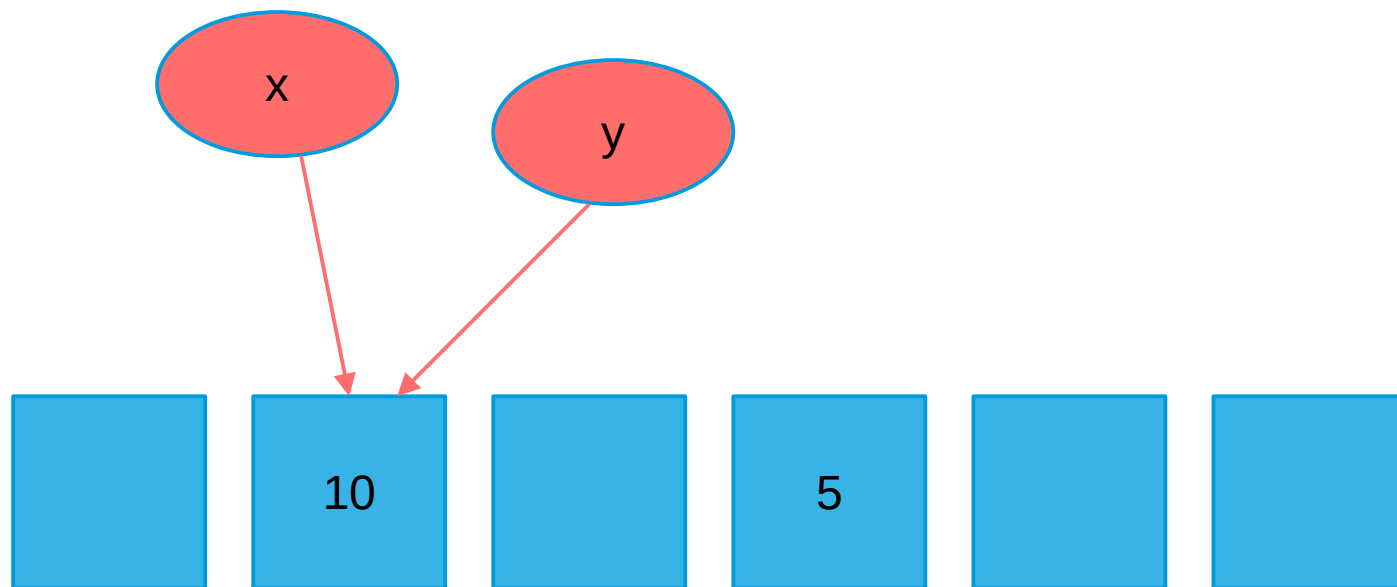
Ячейки памяти



# Переменные - бирки

$y = x$

Ячейки памяти



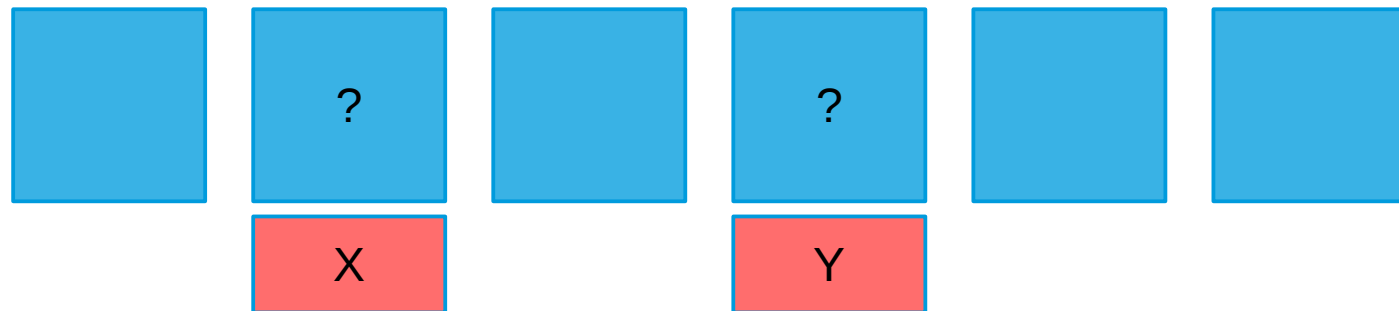


# Переменные - коробки

`int x;`

`int y;`

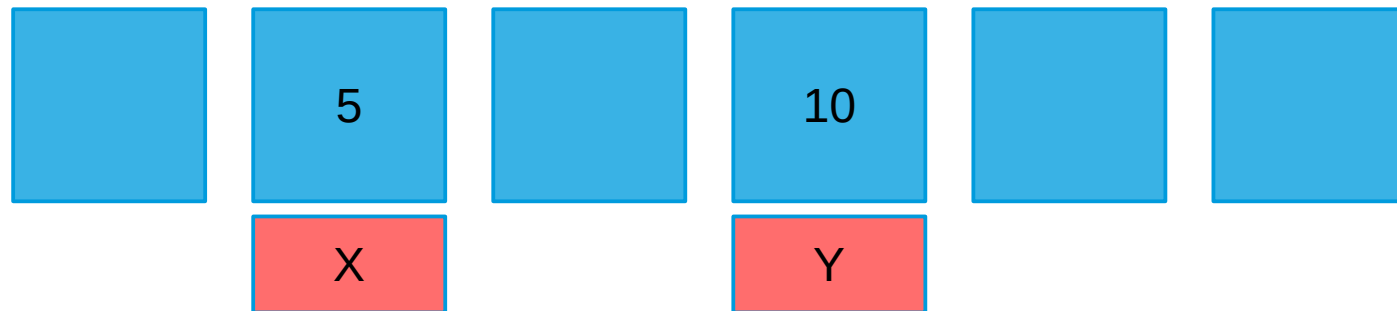
Ячейки памяти



# Переменные - коробки

`x = 5;`  
`y = 10;`

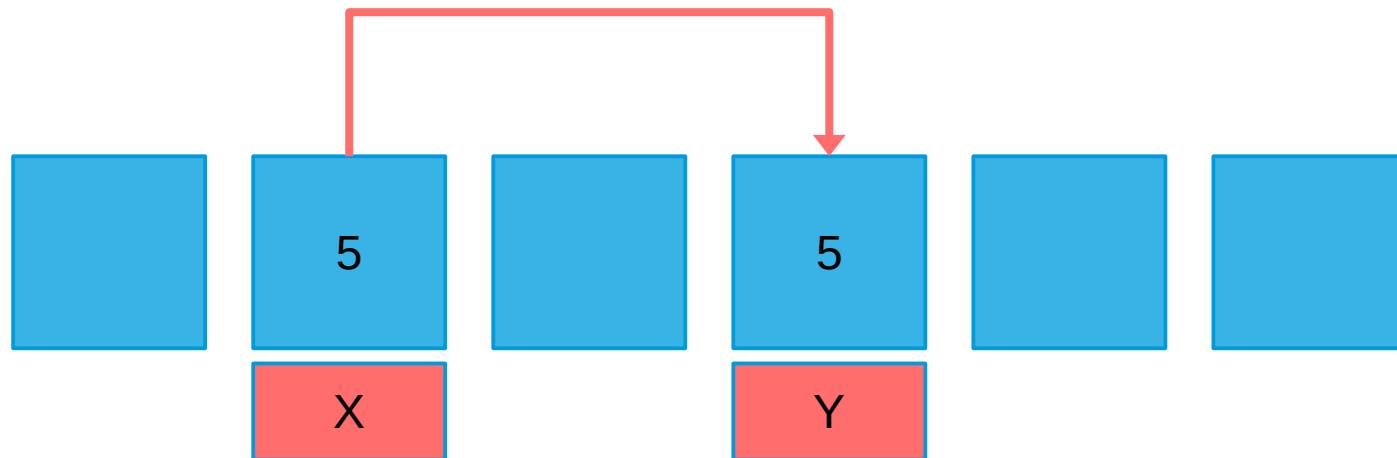
Ячейки памяти



# Переменные - коробки

$$y = x$$

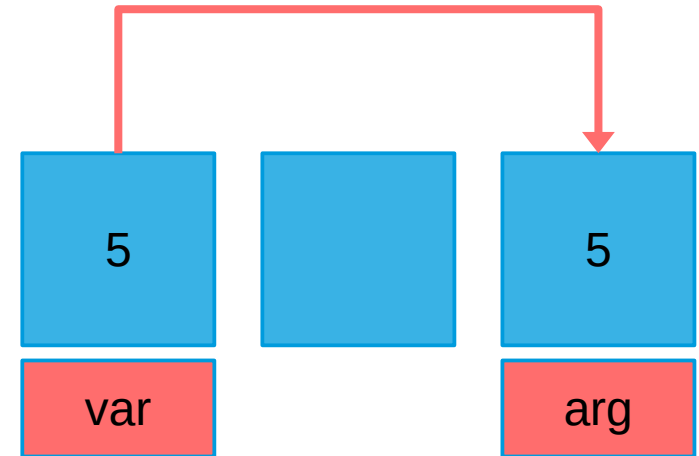
Ячейки памяти



# Передача переменных в функции

```
4 void f(int arg)
5 {
6     printf("arg = %d\n", arg);
7     arg = arg + 100;
8     printf("arg after + = %d\n", arg);
9 }
10
11 int main()
12 {
13     int var = 0;
14     printf("var = %d\n", var);
15
16     f(var);
17
18     printf("var after f = %d\n", var);
19     return 0;
20 }
```

```
<terminated> (exit value: 0) cansat-lex
var = 0
arg = 0
arg after + = 100
var after f = 0
```



# Константы

Константа ничем не отличается от обычной переменной, кроме того, что она должна быть инициализирована при определении и ей нельзя присвоить новое значение.

Чтобы сделать переменную константой нужно использовать модификатор **const**

```
4 int main()  
5 {  
6     >> const int x = 10;  
7     >> const int y; // Ошибка, нужна инициализация  
8  
9     >> x = 20; // ошибка  
10    >> x += 20; // тоже ошибка  
11  
12    >> int a = x; // А вот так можно  
13    >> return 0;  
14 }  
15
```

```
,  
8 void f(const int x)  
9 {  
10    >> x = x + 10; // Ошибка  
11 }  
12
```

# Область видимости и время жизни

Переменные живут и видны от определения до } блока

```
5
6 int main()
7 {
8     printf("x = %d\n", x); // Ошибка, x не определен
9
10    int x = 10;
11    printf("x = %d\n", x); // Теперь все в порядке
12
13    {
14        int y = 100;
15        printf("y = %d\n", y); // Все в порядке
16
17        int x = 20;
18        printf("x = %d\n", x); // Вывод 20. Это уже другой x!
19    }
20
21    printf("y = %d\n", y); // Ошибка! y уже умер
22    printf("x = %d\n", x); // Вывод 10. Это опять старый x
23
24    return 0;
25 }
26
```

# Область видимости и время жизни

Переменные живут и видны от определения до } блока

```
5
6 int main()
7 {
8     printf("x = %d\n", x); // Ошибка, x не определен
9
10    int x = 10;
11    printf("x = %d\n", x); // Теперь все в порядке
12
13    {
14        int y = 100;
15        printf("y = %d\n", y); // Все в порядке
16
17        int x = 20;
18        printf("x = %d\n", x); // Вывод 20. Это уже другой x!
19    }
20
21    printf("y = %d\n", y); // Ошибка! y уже умер
22    printf("x = %d\n", x); // Вывод 10. Это опять старый x
23
24    return 0;
25 }
26
```

# Область видимости и время жизни

Глобальные переменные начинают жить до `main()` и заканчивают жить после неё.

Однако видны так же, только после определения

```
6 void f()
7 {
8     printf("x=%d\n", x); // Ошибка, не видно
9     printf("global_var=%d\n", global_var); // Ошибка, не видно
10 }
11
12
13 int x; // Можно не писать = 0. Это подразумевается и не UB!
14 int global_var = 10;
15
16
17 int main()
18 {
19     printf("x=%d\n", x); // Все в порядке
20     printf("global_var=%d\n", global_var); // Все в порядке
21     return 0;
22 }
```



# Глобальные переменные

Глобальные переменные и функции можно объявлять, а определение писать где-нибудь в другом месте.

Объявлений может быть много  
**Определение должно быть одно!**

```
4
5 // Объявление глобальной переменной
6 extern int x;
7 // Объявление функции
8 void function(int arg);
9
10
11 void f()
12 {
13     printf("x = %d\n", x); // Все ок
14     function(10); // Все тоже ок
15 }
16
17 // Определение переменной
18 int x;
19
20 // Определение функции
21 void function(int arg)
22 {
23     printf("arg = %d\n", arg);
24 }
```

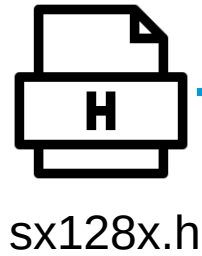
# Статические переменные и функции

Статические функции видны только в родном .c файле. Можно иметь много статических функций с одинаковыми именами в разных файлах.

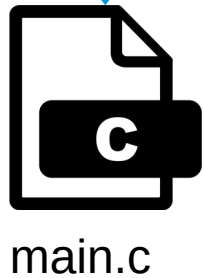
Статические переменные

```
5 static int global; // Видна только в этом файле
6
7 static void f() // Видна только в этом .c файле
8 {
9     static int x = 0; // Живет от инициализации вечно
10    // // // // // Инициализируется один раз
11    printf("x = %d\n");
12    x += 1;
13 }
14
15 int main()
16 {
17     f(); // Выводит 0
18     f(); // Выводит 1
19     f(); // Выводит 2
20     f(); // Выводит 3
21
22     return 0;
23 }
```

# Заголовочные файлы



#include



```
212
213 // Функции для первичной инициализации драйвера
214 // =====
215
216 // Конструктор драйвера
217 int sx126x_drv_ctor(sx126x_drv_t* drv, void* board_ctor_arg);
218
219 // Деструктор драйвера
220 void sx126x_drv_dtor(sx126x_drv_t* drv);
221
222 // Состояние драйвера
223 sx126x_drv_state_t sx126x_drv_state(sx126x_drv_t* drv);
224
225
226 // Функции для первичной инициализации железа, сбросов и standby режимов
227 // =====
228
229 // Сброс чипа через RST пин и состояния драйвера
230 int sx126x_drv_reset(sx126x_drv_t* drv);
231
232 // Базовая конфигурация чипа и драйвера
233 /* Большинство опций, которые используются здесь не поддерживают
234  * повторной реконфигурации. Поэтому эта функция должна использоваться
235  * один раз после резета чипа.
236  * Функция может быть вызвана только в состоянии SX126X_DRV_STATE_STARTUP
237  * И только посредством этой функции можно перейти в первый стэндабай-режим
238  */
239 int sx126x_drv_configure_basic(sx126x_drv_t* drv, const sx126x_drv_basic_config_t* config);
240
241 // Переводит чип в режим standby на RC-цепочке
242 /* Это самый базовый режим в котором отключено все что можно.
```

# Заголовочные файлы

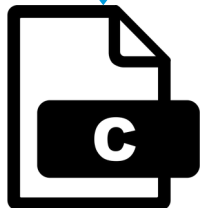


#include

sx128x.h



sx128x.c



main.c

```

sx126x_drv.h  sx126x_drv.c x
457
458 //! Проверка на то, находится ли модуль в standby режиме
459 //! Вынесена отдельно, чтобы часто уж используется *
460 static bool is_in_standby(sx126x_drv_t* drv)
461 {
462     switch (drv->state)
463     {
464     case SX126X_DRV_STATE_STANDBY_RC:
465     case SX126X_DRV_STATE_STANDBY_XOSC:
466     >> return true;
467
468     default:
469     >> break;
470     }
471
472     return false;
473 }
474
475 // =====
476 // =====
477
478
479 int sx126x_drv_ctor(sx126x_drv_t* drv, void* board_ctor_arg)
480 {
481     int rc;
482     memset(drv, 0, sizeof(*drv));
483
484     rc = sx126x_api_ctor(&drv->api, board_ctor_arg);
485     SX126X_RETURN_IF_NONZERO(rc);
486
487     load_defaults(drv);

```

Outline x Build Targets

- sx126x\_drv\_stanaby\_switch\_t : enum sx126x\_drv\_sta
- blocking\_mode\_switch\_call\_t : int(\*) (sx126x\_drv\_t\*, ui
- \$ wait\_busy(sx126x\_drv\_t\*) : int
- \$ set\_antenna(sx126x\_drv\_t\*, sx126x\_antenna\_mode\_t
- \$ get\_time(sx126x\_drv\_t\*, uint32\_t\*) : int
- \$ best\_pa\_coeff\_idx(int8\_t, const sx126x\_pa\_coeffs\_t\*,
- \$ get\_pa\_coeffs(sx126x\_chip\_type\_t, int8\_t, sx126x\_pa
- \$ workaround\_1\_lora\_bw500(sx126x\_drv\_t\*) : int
- \$ workaround\_2\_tx\_power(sx126x\_drv\_t\*) : int
- \$ workaround\_3\_rx\_timeout(sx126x\_drv\_t\*) : int
- \$ workaround\_4\_iq\_polarity(sx126x\_drv\_t\*, bool) : int
- \$ configure\_pa(sx126x\_drv\_t\*, int8\_t, const sx126x\_pa
- \$ set\_rf\_frequency(sx126x\_drv\_t\*, uint32\_t) : int
- \$ set\_lora\_syncword(sx126x\_drv\_t\*, sx126x\_lora\_syncv
- \$ fetch\_clear\_irq(sx126x\_drv\_t\*, uint16\_t\*) : int
- \$ switch\_state\_to\_standby(sx126x\_drv\_t\*, sx126x\_drv
- \$ load\_defaults(sx126x\_drv\_t\*) : void
- \$ is\_in\_standby(sx126x\_drv\_t\*) : bool
- **o sx126x\_drv\_ctor(sx126x\_drv\_t\*, void\*) : int**
- sx126x\_drv\_dtor(sx126x\_drv\_t\*) : void
- sx126x\_drv\_state(sx126x\_drv\_t\*) : sx126x\_drv\_state\_t
- sx126x\_drv\_reset(sx126x\_drv\_t\*) : int
- sx126x\_drv\_configure\_basic(sx126x\_drv\_t\*, const sx1
- sx126x\_drv\_mode\_standby\_rc(sx126x\_drv\_t\*) : int
- sx126x\_drv\_mode\_standby(sx126x\_drv\_t\*) : int
- sx126x\_drv\_configure\_lora\_modem(sx126x\_drv\_t\*, co
- sx126x\_drv\_configure\_lora\_packet(sx126x\_drv\_t\*, con
- sx126x\_drv\_configure\_lora\_cad(sx126x\_drv\_t\*, const
- sx126x\_drv\_configure\_lora\_rx\_timeout(sx126x\_drv\_t\*

# Очень краткий справочник по cstdlib

**<stdbool.h>** - определения для булевых типов (с C99)

**<float.h>** - определения для чисел с плавающей точкой. Минимум, максимум, эпсилон и т. п.

**<stdint.h>** - определения для целых чисел (int8\_t и т. п.)

**<limits.h>** - предельные значения целых чисел

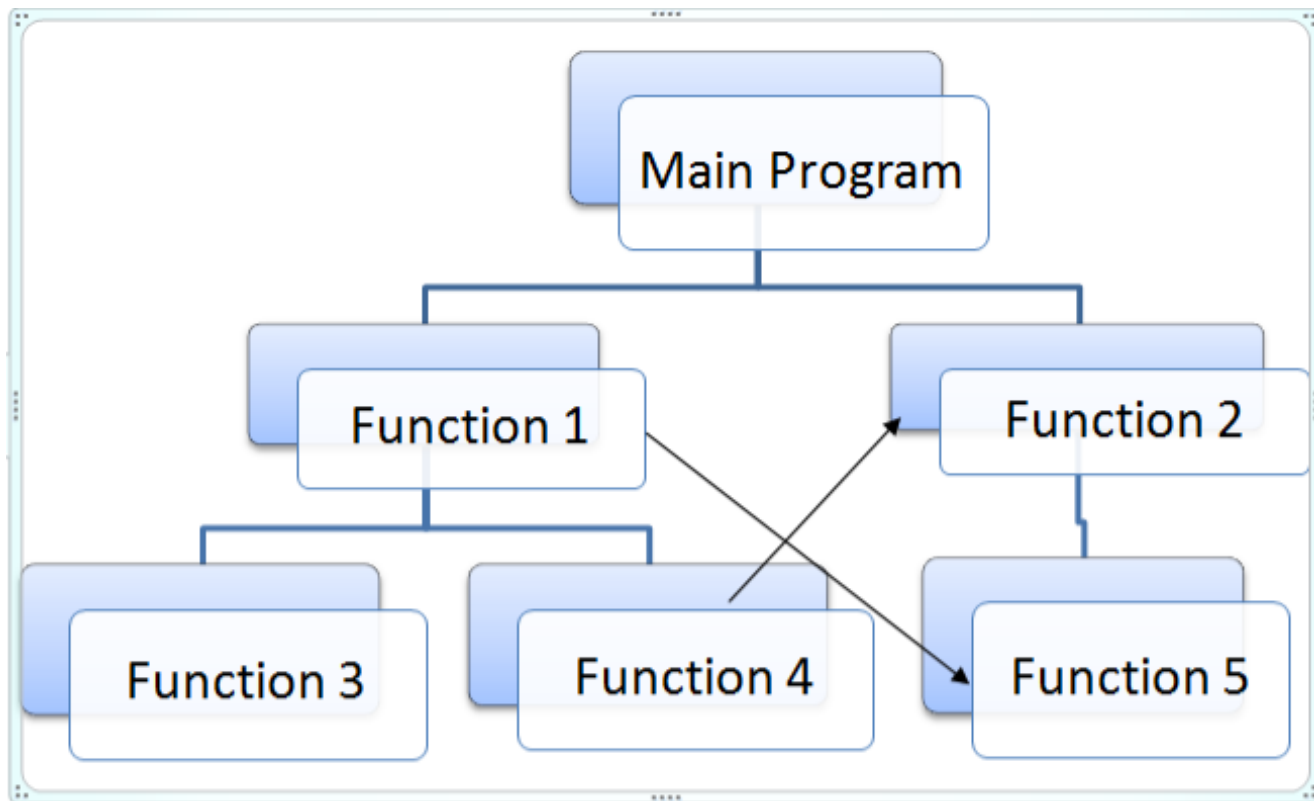
**<math.h>** - математические функции (синус, косинус все такое) и константы (число Пи, число e)

**<stdio.h>** - функции ввода/вывода. Для работы с консолью, файлами и всем таким

**<stdlib.h>** - много разного. Используется для работы с памятью

**<string.h>** - для работы со строками и опять — с памятью

# Процедурное программирование



# Макросы

Помимо функций и переменных, часто используются макросы.

Макросы создаётся при помощи директивы препроцессора **#define**. Они удобны, в случаях, когда Вам нужно использовать какие либо константы, которые могут измениться в процессе разработки.

Макросы могут иметь параметры. Во время работы препроцессора – макросы в тупую заменяются на текст, сопоставленный им, поэтому нужно быть аккуратнее с порядком действий.

```
4 #define PI 3.14159265359
5 #define HEADER_SIZE 10+10+2
6 #define DEG2RAD(arg) arg*PI/180
7
8 int main()
9 {
10     float x = PI;
11     PI = 10; // Бред какой-то
12     int offset = HEADER_SIZE * 2; // Внимание на скобочки
13
14     float radians = DEG2RAD(180);
15     float radians = DEG2RAD(x + 20); // Опять скобочки
16
17     return 0;
18 }
```