Программирование на языке Си

История языка Си

- C (рус. Си) — компилируемый статически типизированный язык программирования общего назначения, разработанный в **1969—1973** годах сотрудником Bell Labs Деннисом Ритчи как развитие языка Би. Первоначально был разработан для реализации операционной системы UNIX, но, впоследствии, был перенесён на множество других платформ. Благодаря близости по скорости выполнения программ, написанных на Си, к языку ассемблера, этот язык получил широкое применение при создании системного программного обеспечения и прикладного программного обеспечения для решения широкого круга задач. Язык программирования C оказал существенное влияние на развитие индустрии программного обеспечения, а его синтаксис стал основой для таких языков программирования, как C++, C#, Java и D.

- По статистике TIOBE до декабря 2015 года Си был самым популярным языком программирования. В декабре на первое место вышла **Java** а Си переместился на второе, при этом считается, что на нем написано порядка 16% всех программ в мире.

Применимость языка Си

- Язык Си – язык низкого уровня абстракции. Это означает, что сущности которыми оперирует язык – очень близки к сущностям, которыми оперирует процессор. Это позволяет очень точно управлять ресурсами ЭВМ (а следовательно использовать их с максимальной эффективностью), но увеличивает сложность написания программ.

- Где применяется язык Си?
- Там где скорость и минимальное потребление памяти превыше всего (Операционные системы, высоконагруженные сервисы, программы для ограниченных по ресурсам систем)
- Там где требуется взаимодействие с железом напрямую (драйверы устройств, различная встраиваемая техника, включая микроконтроллеры)

Специфика языка Си

- Язык С очень похож по синтаксису на многие языки, так как повлиял на их развитие и был в каком-то смысле их предшественником.
- Язык С, как и С++ сохраняет *обратную совместимость* со всеми своими старыми стандартами (весь старый код можно использовать без изменений).
- Язык С – *опасный* язык. Он не запрещает и на защищает от написания совершенной ерунды и совершения логических ошибок в программе.

«Известны 10 преимуществ Паскаля перед Си:) Я приведу только одно, но самое важное:

На Си Вы можете написать:

*for(;P("\n").R-;P("\ "))for(e=3DC;e-;P("_ "+(*u++/8)%2))P("| "+ (*u/4)%2);*

На Паскале Вы НЕ МОЖЕТЕ такого написать»

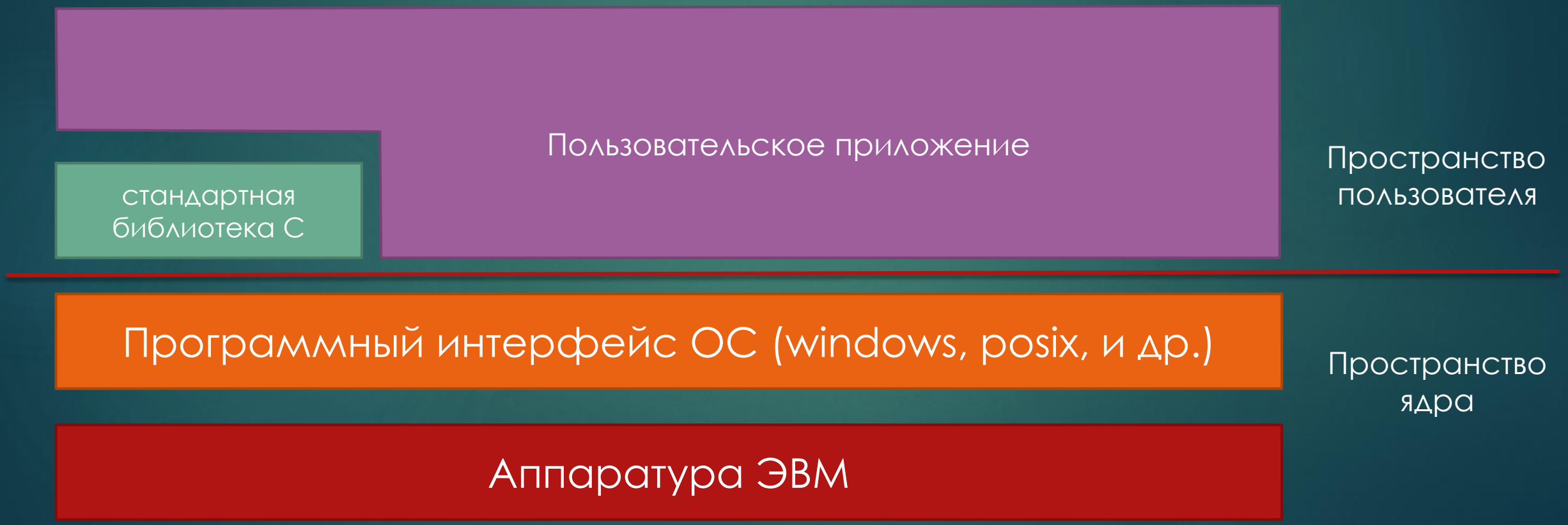
Компилятор, стандартная библиотека и среда разработки

- Для преобразования исходных текстов программ, написанных на языке Си в исполнимые файлы (.exe) необходим набор программ, называемых tool chain (дословно – цепь инструментов) или на жаргоне – компилятор.
- Компиляторов для языка Си очень много. Компилятор, как правило предназначен для конкретной целевой платформы. **Платформа** это совокупность операционной системы и процессора, на котором она работает. Операционная система определяет формат исполняемого файла, а процессор определяет набор машинных инструкций, которыми компилятор может пользоваться при создании машинного кода.
- Например: Windows-x86 (win32); Windows-x86_64 (win64); Linux-x86_64; Android-armeabi; IOS-armv7eabi
- Мы будем использовать тулчейны семейства GNU COMPILLER COLLECTION (**GCC)** как для настольного компьютера так и для микроконтроллеров.

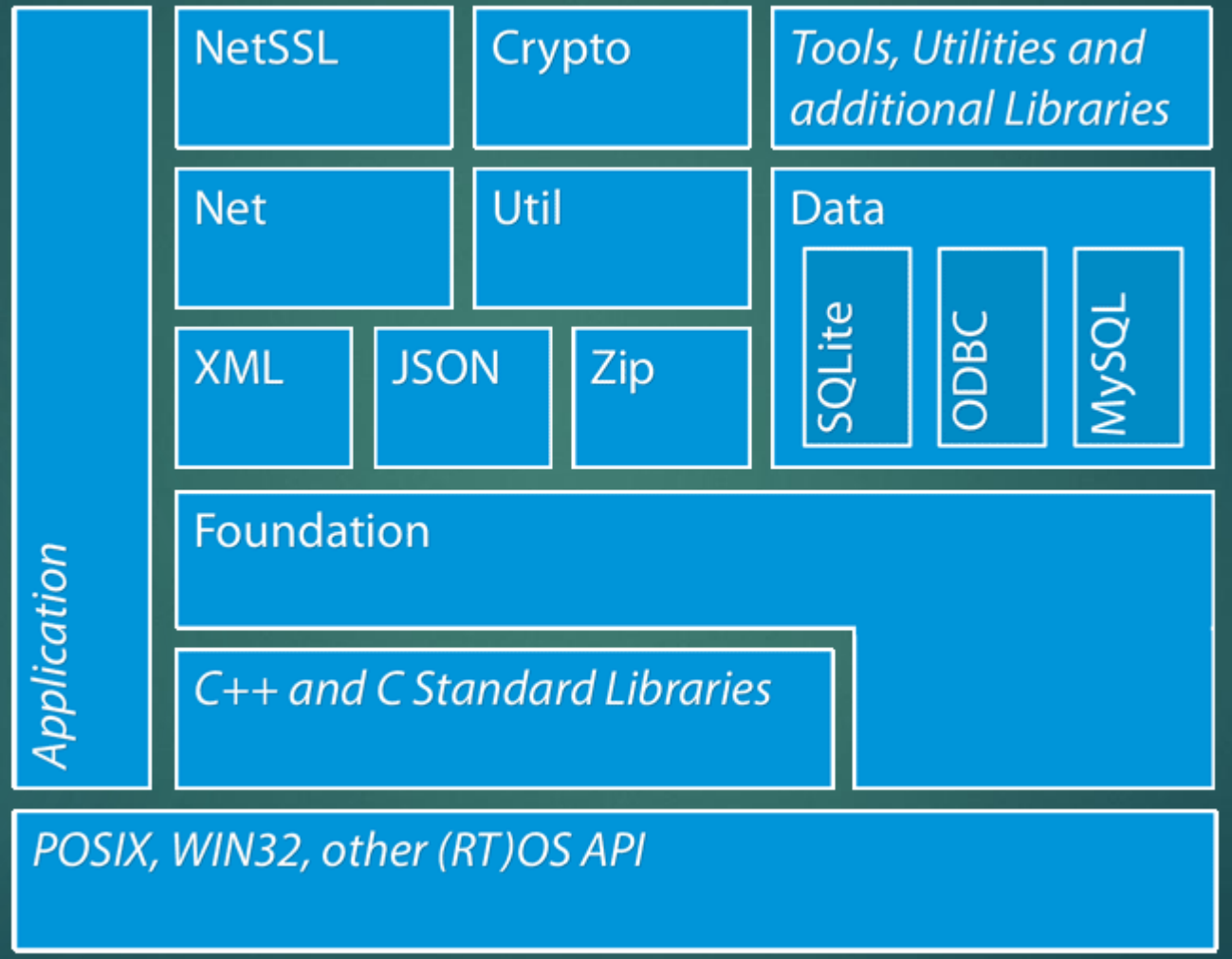
Компилятор, стандартная библиотека и среда разработки

- Каждая операционная система и/или микроконтроллер, предоставляет свой собственный ***программный интерфейс*** для взаимодействия с программой.
- В случае операционной системы – самый простой пример – ввод/вывод данных из консоли.
- Поскольку никому не хочется один и тот же код переписывать для разных операционных систем – существует ***стандартная библиотека*** языка С, в которой в виде некоторых стандартных сущностей определены стандартные интерфейсы взаимодействия с операционной системой и/или с аппаратурой.
- Помимо программных интерфейсов, в стандартную библиотеку включены так же, различные утилитарные сущности – например функции вычисления синуса и пр.
- Стандартная библиотека, как правило, предоставляется вместе с компилятором и ***крайне*** скудна.

Компилятор, стандартная библиотека и системные интерфейсы



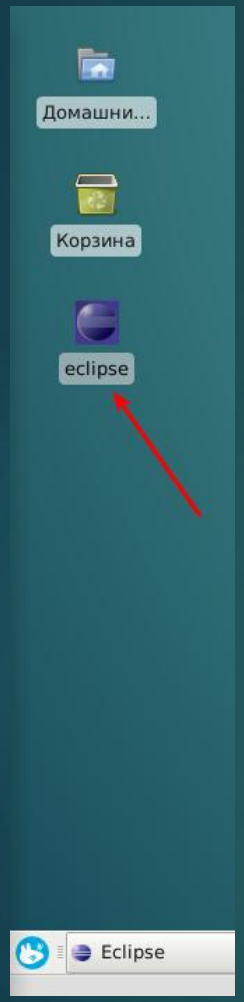
Сторонние библиотеки



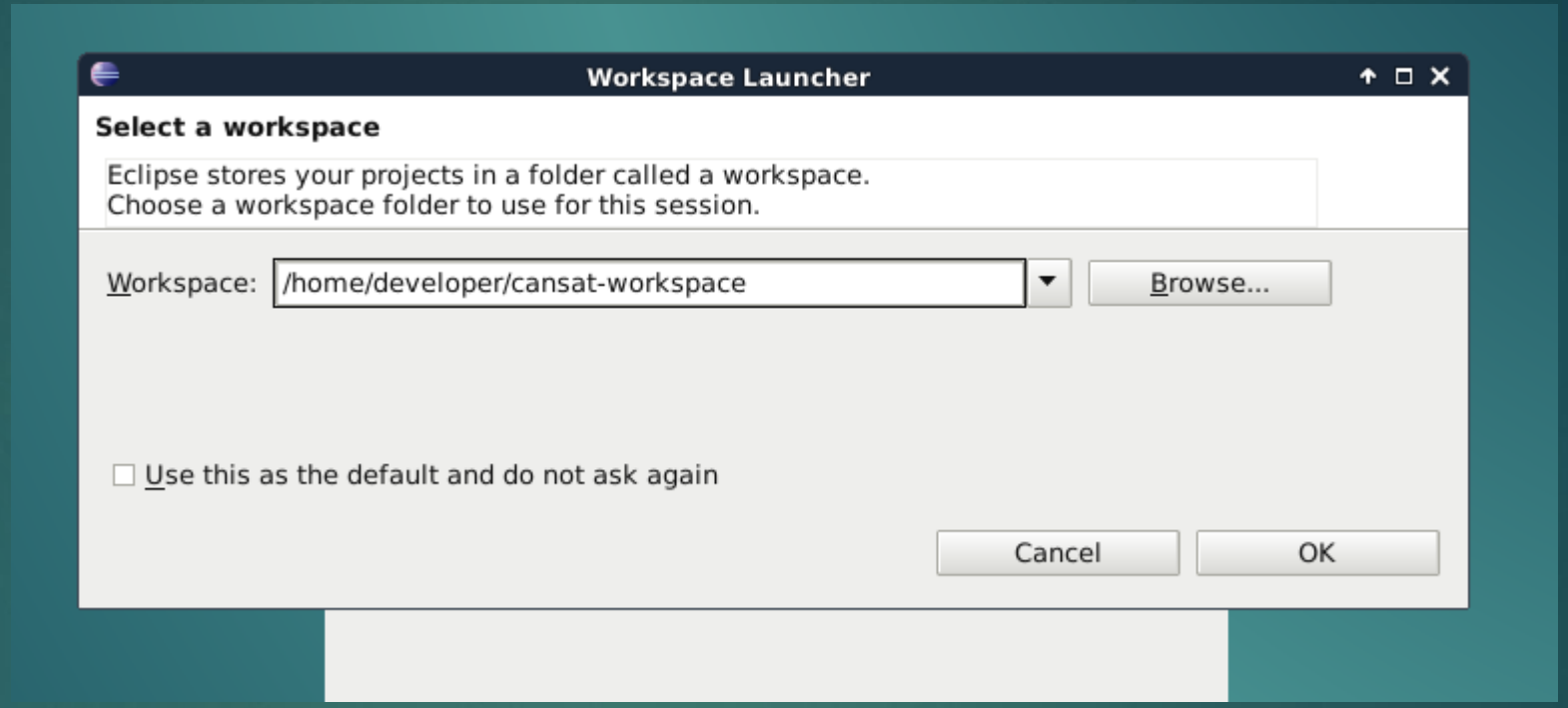
Компилятор, стандартная библиотека и системные интерфейсы

- Тексты программ можно писать хоть в блокноте, а компилятор можно вызывать из командной строки, но это не удобно. Поэтому мы будем использовать ***Интегрированную Среду Разработки (IDE).***
- Сред разработки для языков C/C++, как и компиляторов – огромное множество. Мы будем использовать **eclipse**.
- Eclipse – универсальная среда разработки, которая при помощи плагинов конфигурируется практически под любой современный язык программирования и, в случае C/C++, для любой платформы.

Начнем, пожалуй!

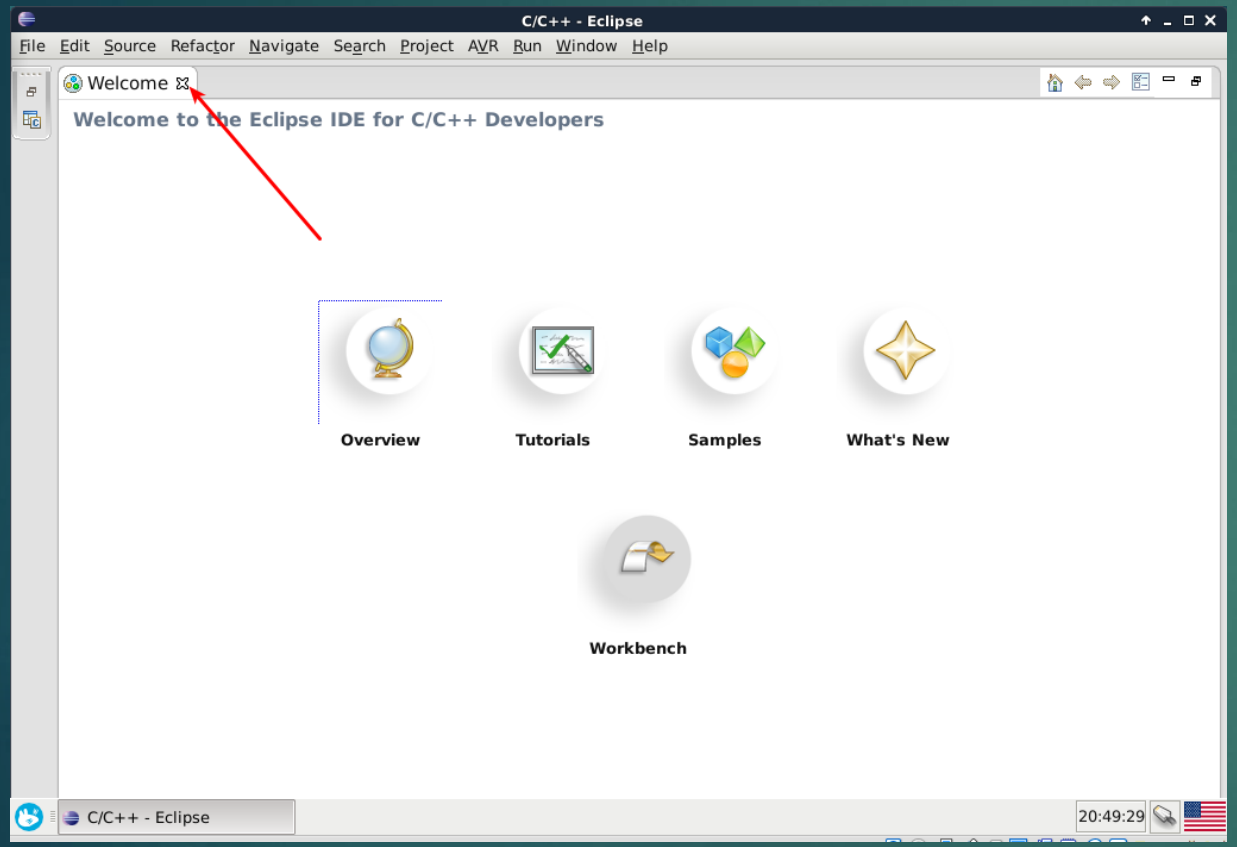


1. Запускаем eclipse

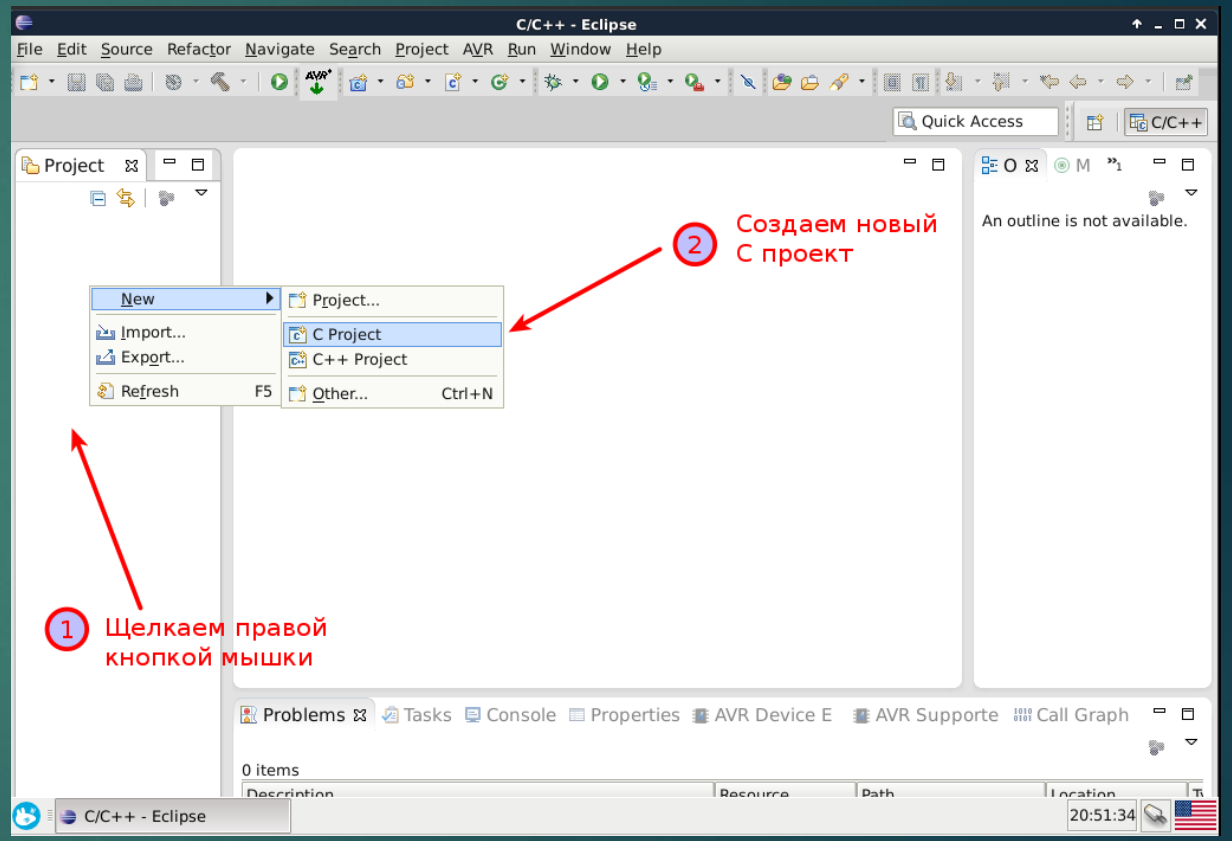


2. Создаем/выбираем workspace

Программа «Hello world»

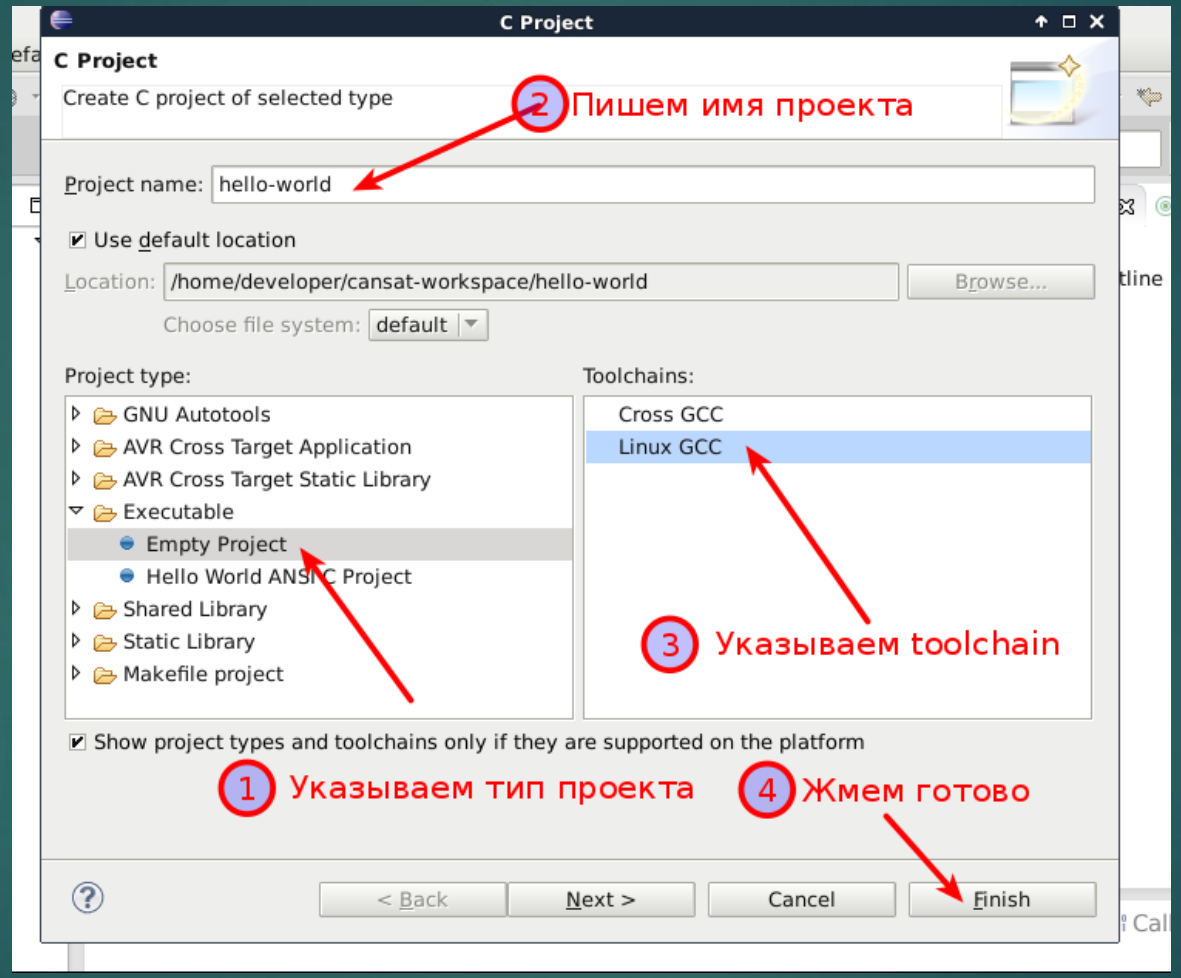


2. Закрываем страницу приветствия



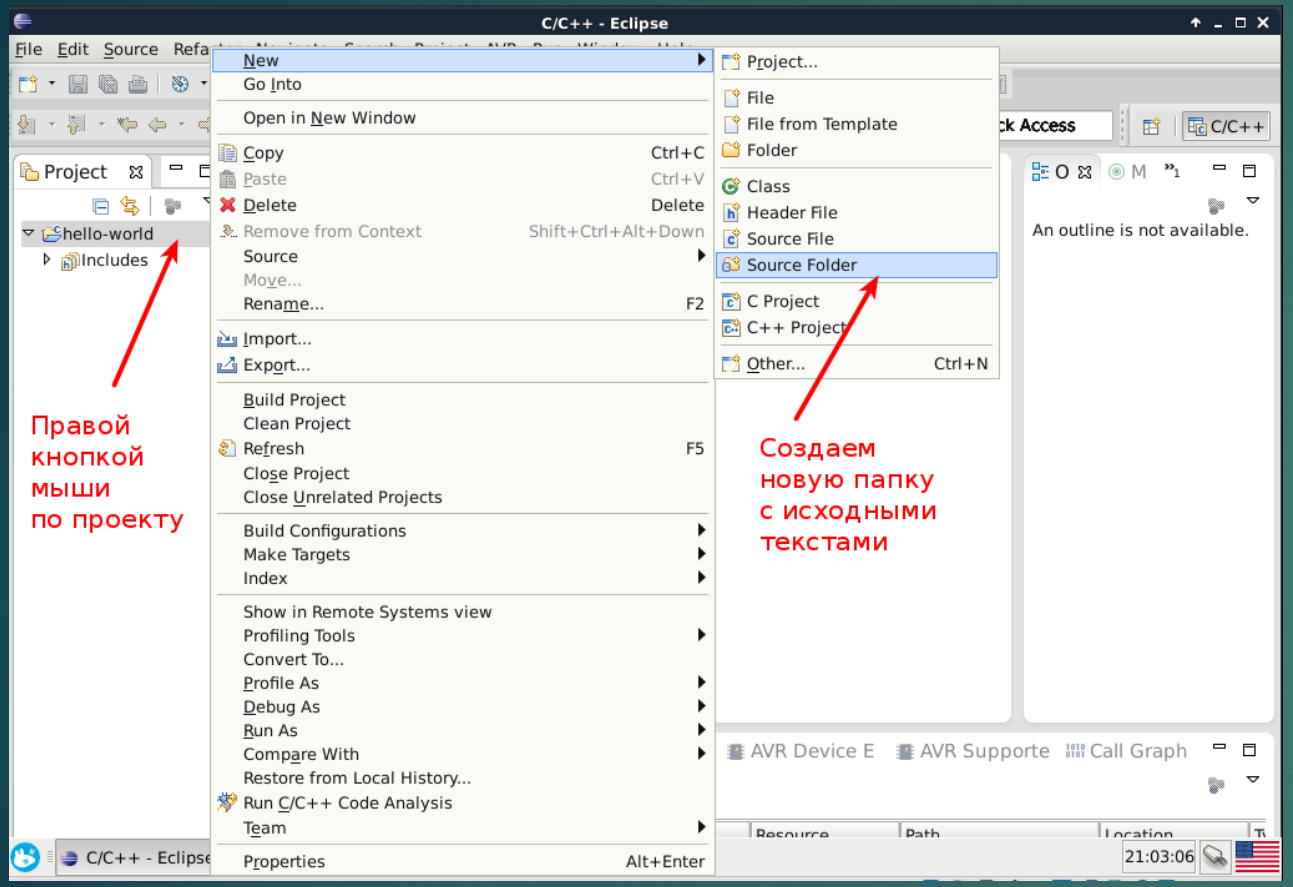
3. Создаем новый С проект

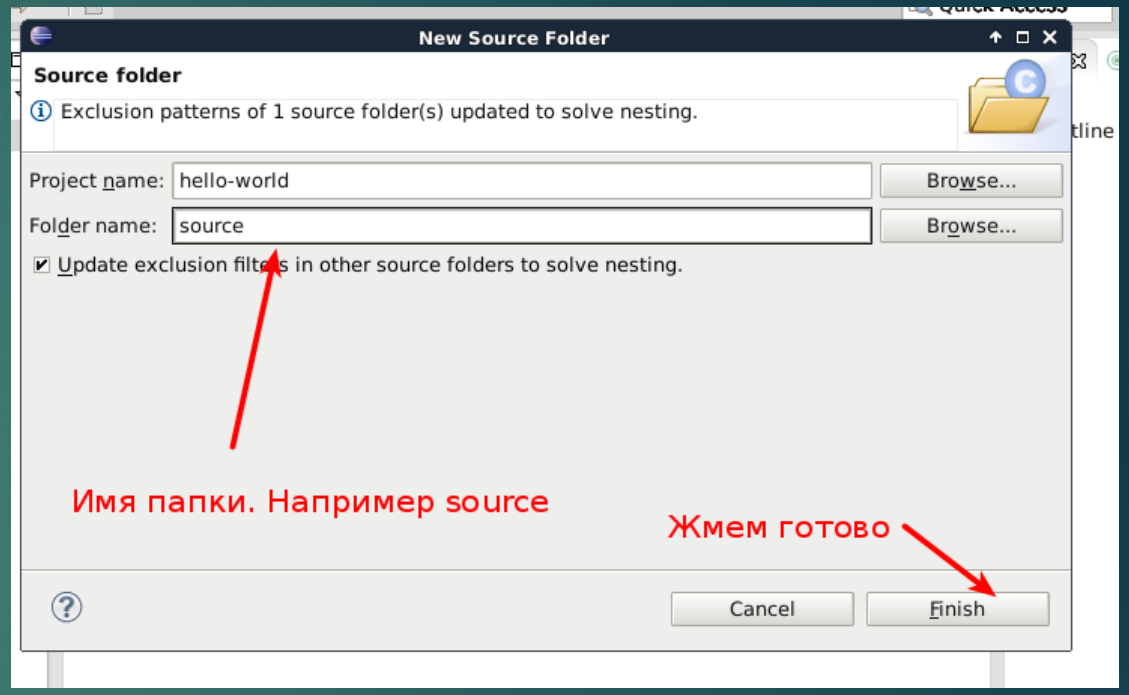
Программа «Hello world»



4. Указываем параметры проекта

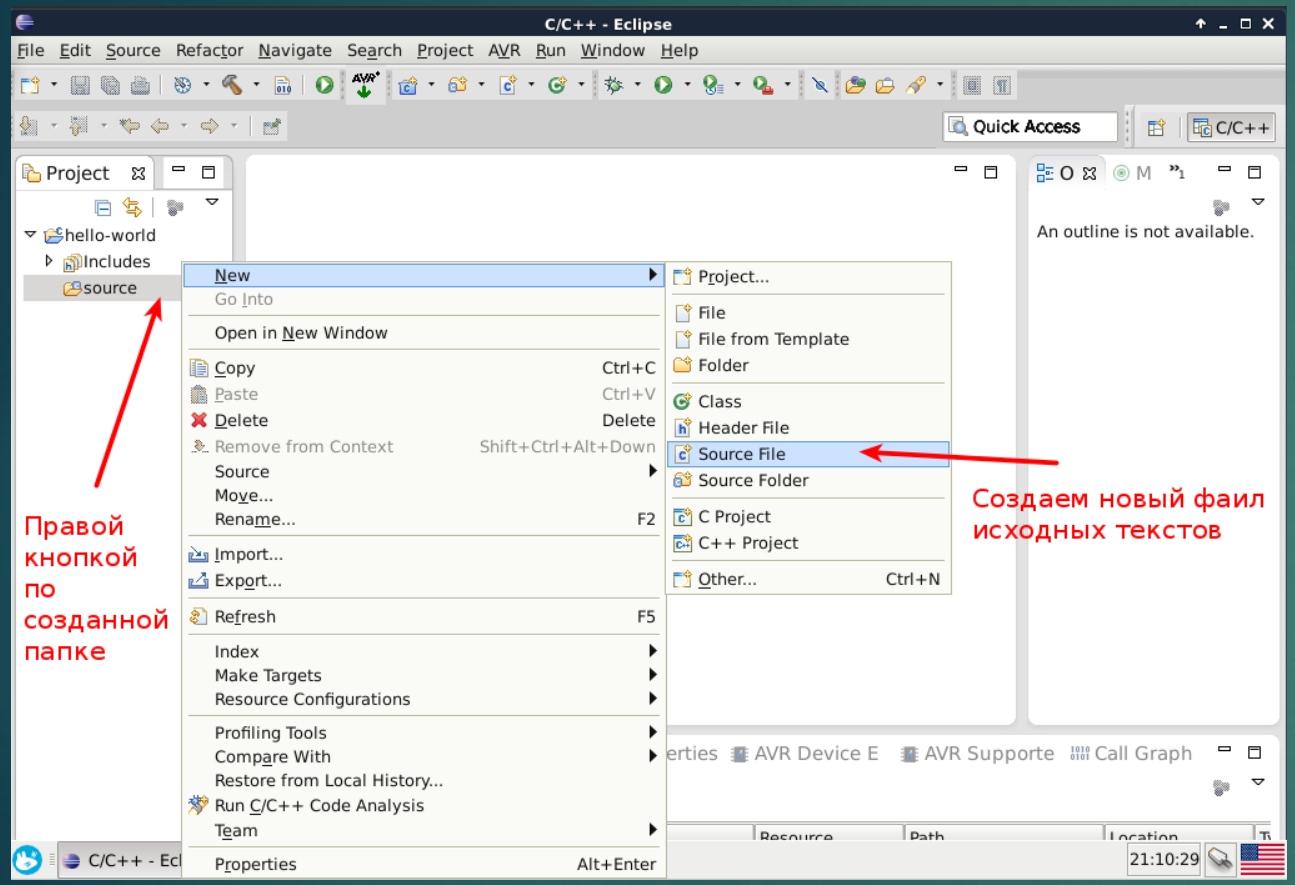
Программа «Hello world»

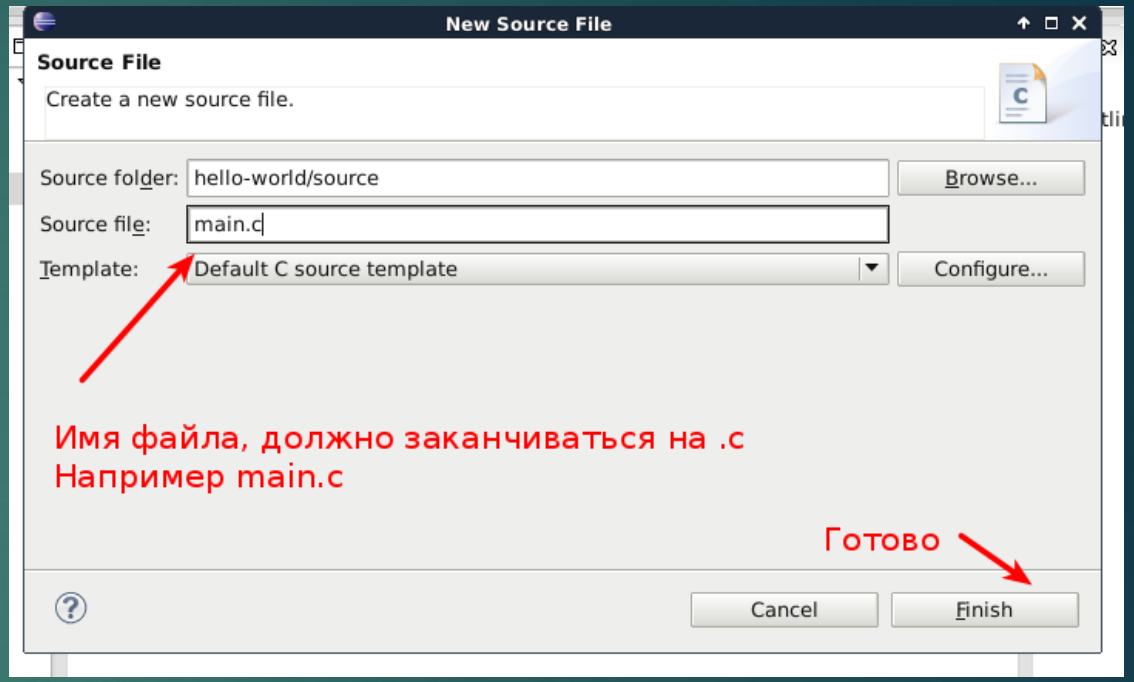




5. Создаем папку для исходных текстов проекта

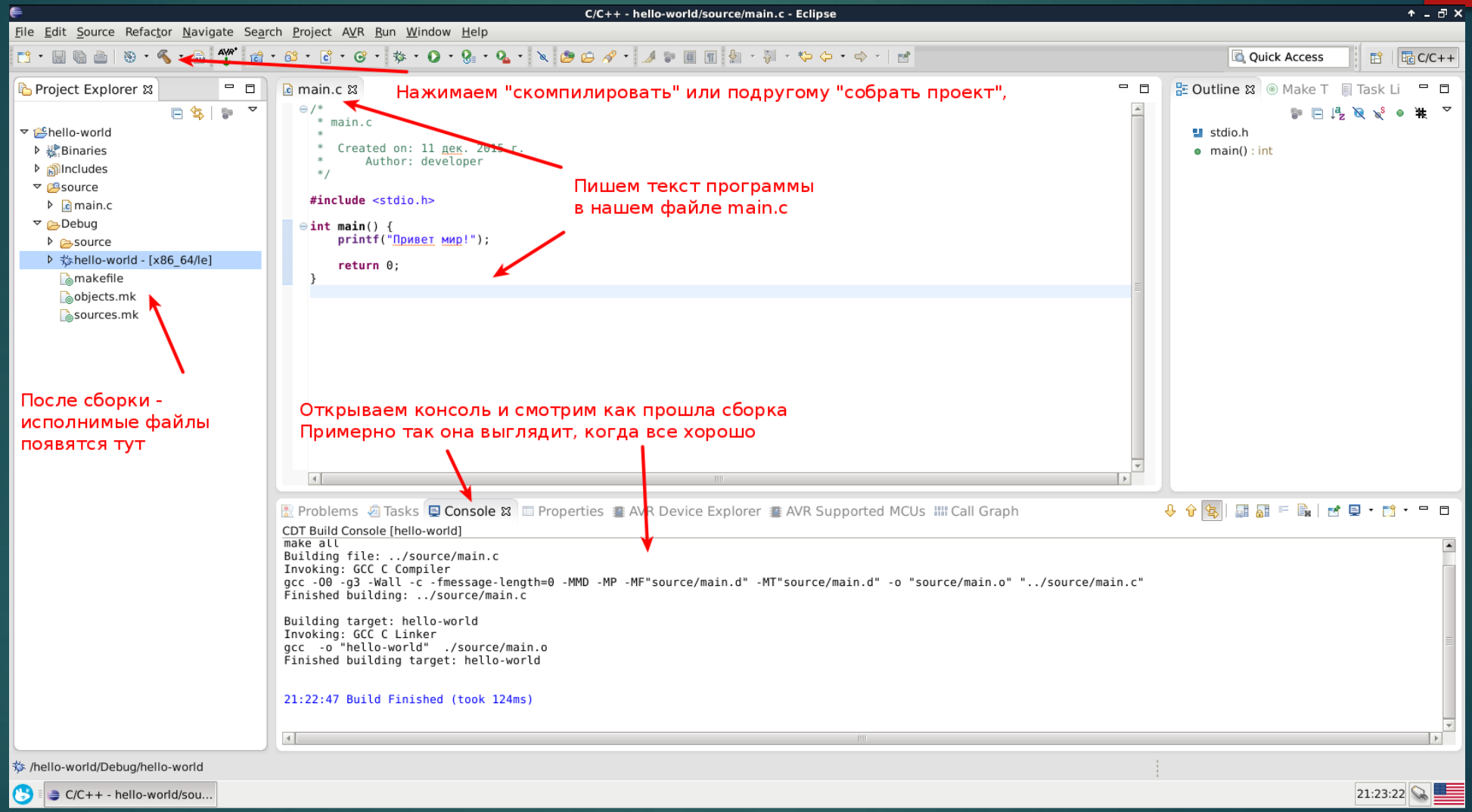
Программа «Hello world»





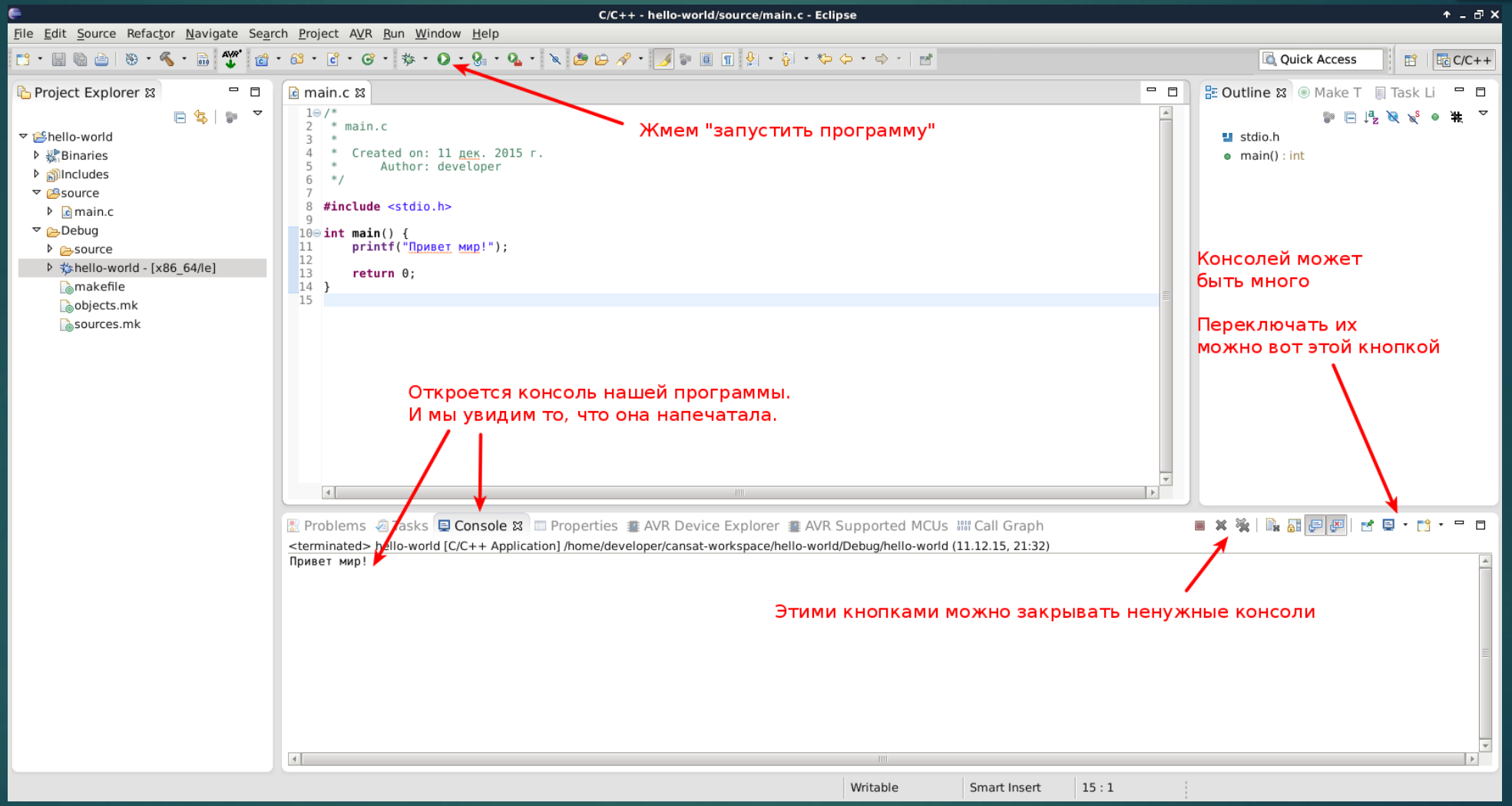
5. создаем файл исходных текстов

Программа «Hello world»



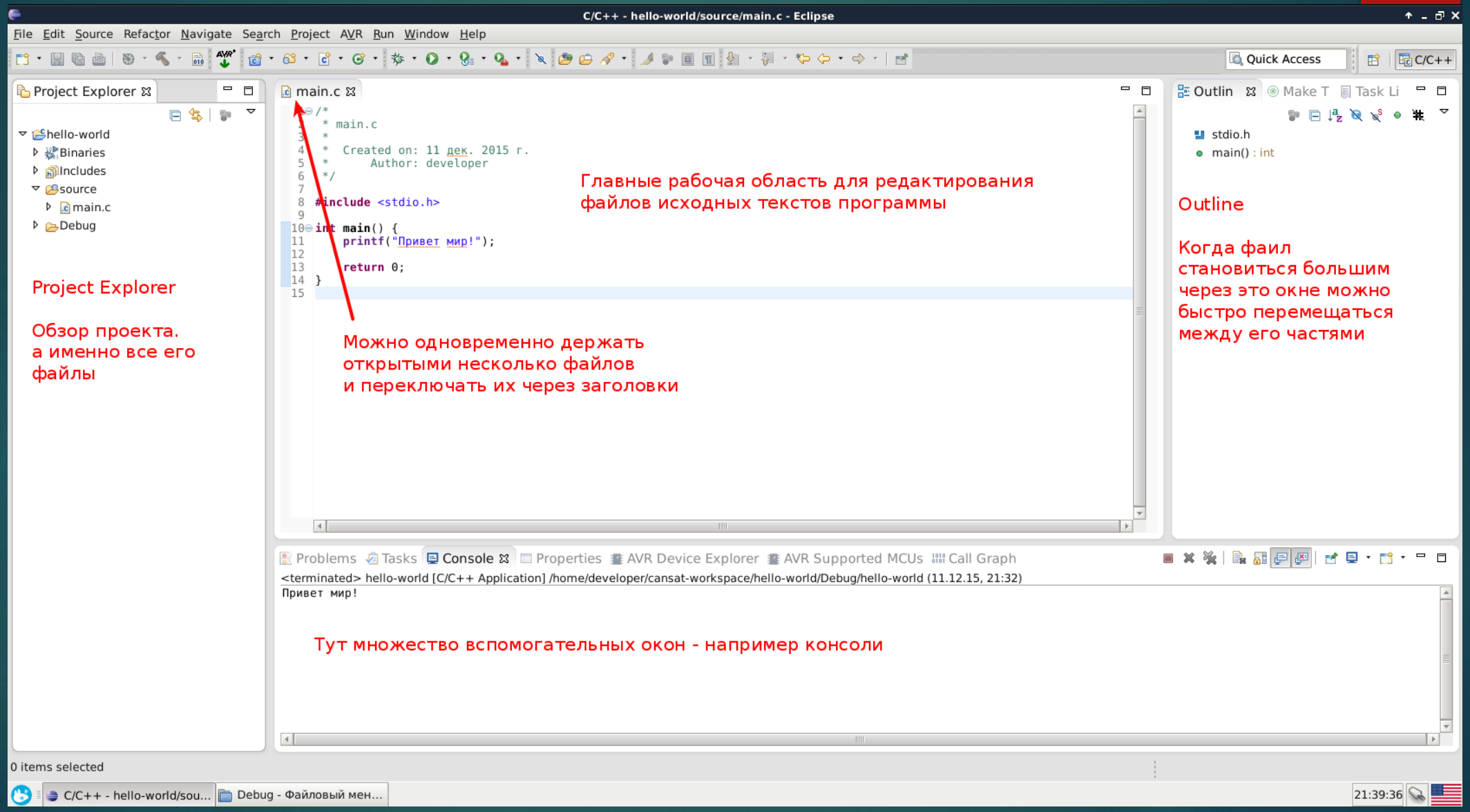
6. Собираем программу

Программа «Hello world»



6. Запускаем программу

Кратко об окнах eclipse



Разбор программы hello world

*main.c

```
1  /*
2   * main.c
3   *
4   *  Created on: 11 дек. 2015 г.
5   *      Author: developer
6   */
7
8  #include <stdio.h>
9
10 int main() {
11     printf("Привет мир!\n");
12
13     return 0;
14 }
15
```

Eclipse выделил зеленым блок комментариев, который Сам же автоматически и создал.

Все, что находиться между /* и */ невидимо для компилятора. Там мы можем писать разные пояснительные тексты для себя или других разработчиков

Кроме /* */ так же компилятор пропускает все от символов // до конца строки

Эклипс будет отмечать комментарии зеленым, поэтому легко понять, если Вы что-то сделали неправильно.

Например:

```
9  // Комментарий
10
11 /* Так тоже комметарий */
12
13 /*
14 И так тоже комментарий
15
16 Тут тоже
17 */
18
19
20 А вот тут уже нет.
21
```

Разбор программы hello world

*main.c

```
1  /*
2   * main.c
3   *
4   *  Created on: 11 дек. 2015 г.
5   *      Author: developer
6   */
7
8  #include <stdio.h>
9
10 int main() {
11     printf("Привет мир!\n");
12
13     return 0;
14 }
15
```

Дальше идет директива препроцессора **#include** (дословно вложить)

Сборка программы - многоэтапный процесс и первый этап – обработка исходников «препроцессором»

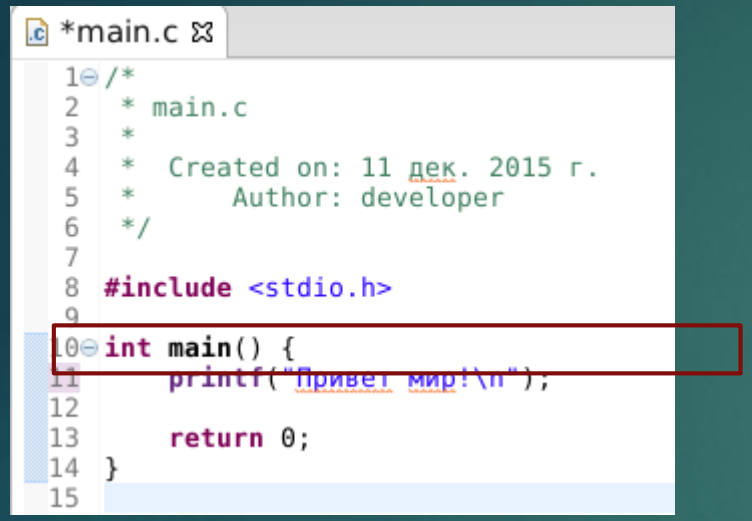
#include – это директива по которой препроцессор – берёт фаил с именем **stdio.h** из стандартных каталогов и включает его содержимое в наш main.c как будто бы мы скопировали весь текст из него и вставили бы вместо директивы

Имя файла указывается в < >, если нужно брать его из стандартных каталогов или в “ ”, если он лежит рядом с нашим main.c

Зачем нам это нужно? **stdio.h** - файл стандартной библиотеки, в котором объявлена функция **printf** Которая выводит сообщение в консоль. Без этого include программа бы не собралась.

stdio означает std + io = standard + input/output. Расширение файла .h означает header, о них позже.

Разбор программы hello world



Следом пошел текст программы а именно – определение функции main. Программы на языке Си всегда состоят из переменных и функций. Функций в программе может быть сколько угодно – как стандартных (та же **printf)**, так и пользовательских – которые мы пишем сами (например наша **main)**. MAIN – особенная функция. С нее начинается и ею же заканчивается выполнение программы.

Функции описываются вот так
тип_возвращаемого_значения имя_функции ([список аргументов])
{
Тело функции
}

Получается, что наша функция возвращает значение типа int, имеет название main и не имеет никаких аргументов – ()

Тело функции

*main.c

```c
/*
 * main.c
 *
 *  Created on: 11 дек. 2015 г.
 *      Author: developer
 */

#include <stdio.h>

int main() {
    printf("Привет мир!\n");

    return 0;
}
```

В теле функции описано то, что она делает – для чего она собственно и пишется.

Наша функция main делает две простых вещи
)1вызывает другую функцию – **printf**, передавая ей в аргументе текст *Привет мир!\n* и сразу же завершается возвращая значение 0.

Обратите внимание, что каждая операция завершается точкой с запятой. В си – это обязательное требование.

```c
#include <stdio.h>


int main() {
    printf("Привет мир!\n");

    return 0;
    printf("Эта операция не выполнится");
}
```

Значение, которое возвращает функция, задается оператором **return**. В формате: **return то_что_нужно_вернуть ;**

Помимо указания возвращаемого значения – return завершает функцию и все, что идет после него – не будет выполнено. Поэтому return это не только вернуть какое-то значение, но и вернуть управление процессом выполнение программы на уровень выше

Вызов функции

Для вызова функции, нужно написать её имя и в скобочках аргументы

Например:

```c
#include <stdio.h>

void print_number(int number) {
    printf("Привет %d\n", number);
}

int main() {
    print_number(10);
    print_number(20);
    print_number(30);

    return 0;
}
```

Вывод:

Привет 10

Привет 20

Привет 30

Главную функцию main вызывает операционная система. Ей же мы возвращаем то самое возвращаемое значение (return 0 в конце). Когда функция main завершает – завершается и вся наша программа.

Обратите внимание, что при вызове функции – нужно обязательно передавать ей все требуемые аргументы

```c
#include <stdio.h>

void print_number(int number)
{
    printf("привет %d\n", number);
}

int main()
{
    print_number(); // так нельзя
    print_number(10, 20, 30); // так тоже нельзя

    return 0;
}
```

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 нетекстовые значения, которые так же нужно выводить на печать.

Базовые спецификаторы функции printf

Код	Формат
%c	Символ
%d	Десятичное целое число со знаком
%e	Экспоненциальное представление числа (в виде мантиссы и порядка) (e 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. Подробнее написано, например, на вики: https://ru.wikipedia.org/wiki/Printf

Тип «НИЧЕГО»

void – дословно пустота. Означает отсутствие значения ну и логично, что не может хранить ничего.

Используется как тип возвращаемого значения функции, которой не нужно возвращать ничего

```
11 void deployParachute() {
12     // какие-то операции по сбросу парашюта
13
14     return; // пустой ретурн, без значения, его можно опустить
15     // как обычно - ничего после ретурна не выполняется
16 }
17
```

Целочисленные типы

Целочисленные типы – типы, способные хранить целые числа

char
(1 байт == 8 бит)
signed: [-128, +127], unsigned: [0, +255]

short int, short, int
(>= 2 байта == 16 бит)
signed: [-32767, +32767], unsigned: [0, +65535]

long int, long
(>= 4 байта == 32 бит)
signed: [-2147483647, +2147483647], unsigned: [0, +4294967295]

long long int, long long
(>= 8 байт == 64 бит)
[-9223372036854775807, +9223372036854775807], unsigned: [0, +18446744073709551615]

Целочисленные типы

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]

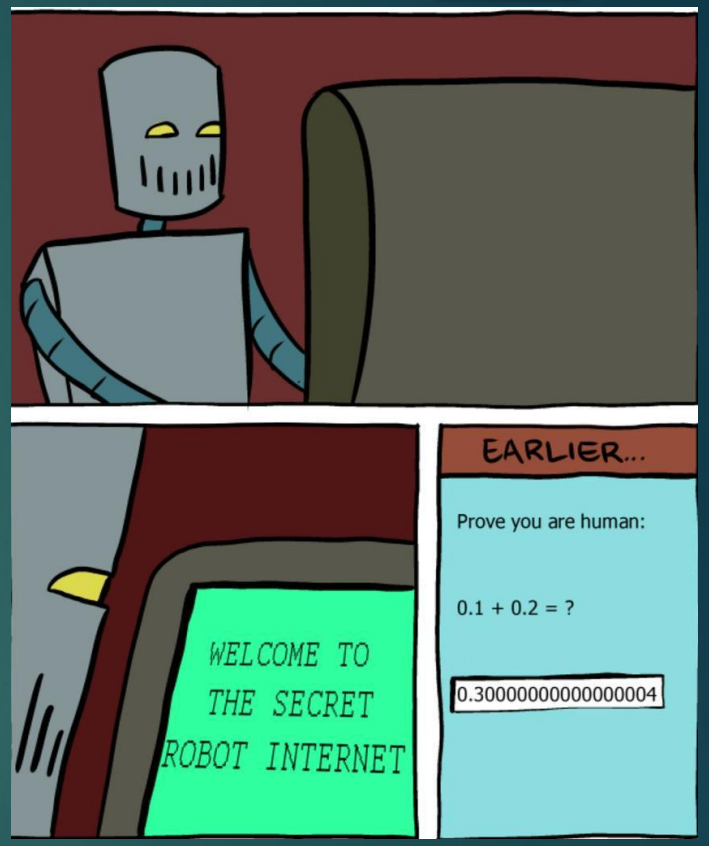
Так же может быть полезен файл **limits.h** (#include <limits.h>) в нем определены минимальные/максимальные значения целочисленных типов в макросах **SHRT_MIN, SHRT_MAX, USHRT_MIN, USHRT_MAX** и прочие.

Типы с плавающей точкой

Их всего два

float
4 байта (32 бита)
$[-3.4*10^{38}, +3.4*10^{38}]$
примерно 7 знаков точности

double
8 байт (64 бита)
$[-1.7*10^{308}, +1.7*10^{308}]$
Примерно 15 знаков точности.



Имеют файл со вспомогательными макросами **<float.h>**

Переменные

Рассмотрим такой пример – допустим, у нас есть функция, которая возвращает напряжение на датчике давления. Нам нужно посчитать давление в Паскалях из этого напряжения и вывести на экран.

Формула для вычисления давления из напряжения –
pressure = mV * (1000 - 100) + 100

```
// Функция, которая возвращает нам напряжение на датчике
int getPressureSensorMillivolts() {
    // пока просто вовзращает 10ку всегда.
    return 10;
}


int main() {
    // определение переменной rawVoltage
    int rawVoltage;
    // присваивание ей значения, возвращенного функцией.
    rawVoltage = getPressureSensorVoltage();

    // определение другой переменной и одновременная
    // инициализация её значениием
    // давления, вычисленным по формуле
    int pressure = rawVoltage * (1000 - 100) + 100;

    // вывод значения на экран
    printf("Давление: %d Па", pressure);

    // завершение программы
    return 0;
}
```

Теперь наш main начинается с **определения** переменной.

Переменные – это хранилище данных определенного 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 указывать не обязательно

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!

Операции с переменными

Преобразование типа

```
uint8_t x = 10;
Int16_t y = x; // неявное преобразование типа, все в порядке

Int16_t x = 1000;
Int8_t y = x; // переполнение. y не способен хранить в себе 1000 и сохранит остаток от 1000/256
// В этом  случае компилятор Вас предупредит, но если вы знаете что делаете, x например == 1 и переполнения не будет, можно использовать явное преобразование

y = (int8_t)x;
```

Аналогично

```
uint16_t x = 65500;
Int16_t y = x; // снова переполнение, знаковый Y способен хранить максимум 32767;

Int16_t x = -100;
uint16_t y = x; // тоже ничего хорошего не выйдет. Беззнаковая переменная не хранит отрицательные числа.
```

Операции с переменными

Инкремент – декремент

```
x++; ++x; // аналогично x = x + 1;
x--;   --x; // аналогично x = x – 1;

Int x = 1;
Int y = x++;
// В итоге y = 1; x = 2;

Int x = 1;
Int y = ++x;
// В итоге y = 2; x = 2;
```

Операции с переменными

Логические операции

```
Int x = 10;

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

Не стоит сравнивать знаковые и без знаковые числа.

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**, 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;
```

Константы

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

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

```
2  int main() {
3
4      const int x = 10;
5      const int y; // ошибка, константа должна быть инициализированна
6
7      x = 20;  // ошибка, нельзя изменять константу после инициализации
8      x += 20; // тоже ошибка
9
10     int a = x; // Можно читать значение константы
11     return 0;
12 }
13
```

Передача переменных в функции и возвращаемые значения

```
int main()
{
    int a = 10;

    // Переменные отлично передаются в функции
    int result = sum(a, 50);

    // Возвращаемое значение функции - это тоже значение. Его можно присваивать
    // переменной или отдавать сразу же другой функции
    printf("result = %d\n", sum(15, 15));

    // Возвращаемое значение функции можно и не забирать вовсе
    sum(a, 50);

    // В качестве аргументов можно давать переменные других типов,
    // которые неявно приводятся к нужным.
    char x = 15;
    sum(x, x);

    // Или явно приводятся
    int64_t y;
    sum((int)y, (int)y);

    return 0;
}
```

Особенности передачи аргументов в функции

Переменные передаются в функции «по значению». Передается лишь значение аргумента, но не он сам.

```c
#include <stdio.h>


void argModificationTest(int arg) {
    printf("Аргумент в функции %d", arg);
    arg = arg + 10;
    printf("Аргумент в функции после изменения %d", arg);
}


int main() {
    int x = 0;
    printf("Аргумент в main %d", x);
    argModificationTest(x);
    printf("Аргумент в main после вызова функции: %d", x);

    return 0;
}

```

Вывод:

Аргумент в main 0

Аргумент в функции 0

Аргумент в функции после изменения 10

Аргумент в main после вызова функции: 0

Ветвление программы. “IF”

```
if (условие) {
    если_условие_верно;
} else {
    если_условие_неверно;
}
```

Например:

```c
#include <stdio.h>

int main() {
    int x = 10;

    if (x > 5)
    {
        printf("X больше пяти");
    }
    else
    {
        printf("X не больше пяти");
    }

    return 0;
}
```

else можно опустить

```c
#include <stdbool.h>
#include <stdio.h>

bool something_happened() { return true; }

int main()
{
    if (something_happened())
        printf("что-то случилось!");

    return 0;
}
```

Если оператор всего один – можно опустить скобки {}

```c
    int x;

    if (x > 5)
        printf("x больше пяти");
    else if (x < 5)
        printf("x меньше пяти");
    else
        printf("x == 5!");

```

Ветвление программы. “CASE”

```
12
13 // Вывод на печать названия дня недели по его номеру
14 void print_day_of_the_week(int day_number)
15 {
16     switch (day_number)
17     {
18         case 1:
19             printf("понедельник");
20             break;
21
22         case 2:
23             printf("вторник");
24             break;
25
26         case 3:
27             printf("среда");
28             break;
29
30         // и так далее
31
32         default:
33             printf("Неверный номер дня недели: %d", day_number);
34             break;
35     };
36 }
37
```

Работает только на строгое равенство
Нельзя писать “**case > 5**”

```
12
13 // Проверка - выходной ли сегодня
14 bool is_free_day(int day_number)
15 {
16     switch (day_number)
17     {
18         case 1: case 2: case 3:
19         case 4: case 5:
20             printf("сегодня рабочий день");
21             return false;
22             // break опущен сознательно
23             // так как return и так не пустит нас дальше
24
25         case 6: case 7:
26             printf("Сегодня выходной!");
27             return true;
28
29         default:
30             printf("Неверный номер дня недели: %d", day_number);
31             return true;
32     };
33 }
34
```

Область видимости переменных

- Переменные видны от их определения, до закрывающей скобочки блока { }, в котором они определены

```
int main() {

    printf("y = %d", y); // Ошибка: y еще не определена
    int y;

    {
        int x = 20;
        printf("y = %d", y); // все в порядке
    }

    printf("x = %d", x); // ошибка - x здесь не видна
    printf("y = %d", y); // все в порядке

    return 0;
}
```

Область видимости переменных

- Имена переменных можно перекрывать на более глубоких вложенностях

```
int main() {

    int x;
    x = 0;

    {
        int x;
        x = 20;

        printf("x = %d", x);
    }

    printf("x = %d", x);

    return 0;
}
```

Какой будет вывод?

Область видимости переменных

- Переменные определенные внутри функций или на более глубоких уровнях вложения блоков { } называеются **локальными**.
- Переменные опередеденные на самом первом уровне вложеннсти (вообще вне { } называются **глобальными.** Глобальные переменные видны от их определения, до конца файла.

```
// опредееление глобальной переменной
int globalX = 10;


void printGlobalX() {
    printf("globalX в функции = %d", globalX);
}

int main() {

    printf("globalX в main = %d", globalX);

    globalX += 10;
    printf("globalX в main = %d", globalX);

    globalX += 10;
    printGlobalX();

    return 0;
}
```

globalX в main = 10
globalX в main = 20
globalX в функции = 30

Область видимости функций

Подобно глобальным переменным функции видны от их определения до конца файла

```c
#include <stdio.h>

void f1() {
    printf("Вызвана функция f1");
}


void f2() {
    f1();
    printf("Вызвана функция f2");
}


void f3() {
    f2();
    printf("Вызвана функция f3");
}


int main() {
    f3();

    return 0;
}
```

Макросы

- Помимо функций и переменных, часто используются макросы. Макросы создается при помощи директивы препроцессора **#define.** Они удобны, в случаях, когда Вам нужно использовать какие либо константы, которые могут измениться.
- Например **#define CALIBRATED_PRESSURE_COEFFICIENT 12**

- Макросы могут иметь параметры.
  #define DEG_TO_RAG(degrees) degrees*3.14/180
- Во время работы препроцессора – макросы заменяются на текст, сопоставленный им, поэтому нужно быть аккуратнее с порядком действий.

Массивы и циклы

- Представим, что нам нужно сделать несколько измерений температуры и посчитать её среднее значение. Можно сделать так:

```c
#include <stdio.h>
#include <stdint.h>

uint8_t getTemperature() {
    uint8_t tempValue;
    // ...
    // операции с датчиком
    // ...
    return tempValue;
}

int main() {
    uint8_t temp0 = getTemperature();
    uint8_t temp1 = getTemperature();
    uint8_t temp2 = getTemperature();
    uint8_t temp3 = getTemperature();
    uint8_t temp4 = getTemperature();

    uint8_t averageTemp = (temp0 + temp1 + temp2 + temp3 + temp4);
    averageTemp /= 5;

    printf("Среднее значение температуры: %d", averageTemp);
    return 0;
}
```

Массивы и циклы

- Гораздо лучше будет использовать массив и цикл

```c
#include <stdio.h>
#include <stdint.h>

#define TEMP_ARRAY_SIZE 500

uint8_t getTemperature() {
    uint8_t tempValue;
    // ...
    // операции с датчиком
    // ...
    return tempValue;
}

int main() {
    uint8_t tempValues[TEMP_ARRAY_SIZE];

    for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
        tempValues[i] = getTemperature();
    }

    uint8_t averageTemp = 0;
    for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
        averageTemp += tempValues[i];
    }

    averageTemp /= TEMP_ARRAY_SIZE;
    printf("Среднее значение температуры: %d", averageTemp);
    return 0;
}
```

Массив – это набор переменных Одного и того же типа. Объявляется как Переменная, но в конце указывается размер массива в скобочках [ ].

Можно использовать модификаторы еременных (например **const**).

Для доступа к элементу массива нужно написать его имя и затем в скобочках [ ] номер элемента. Нумерация с нуля.

Массивы и циклы

Многомерные массивы и инициализация массивов

```
int main() {

    // многомерные массивы
    int twoDimensionalArray[5][10];
    int threeDimensionalArray[5][10][15];

    // доступ к элементу
    printf("arrayValue: %d", twoDimensionalArray[0][0]);
    printf("arrayValue: %d", threeDimensionalArray[0][0][0]);

    // инцилазиация массивов
    int array1[10]; // в элементах массива что угодно;
    int array2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // все элементы массива указаны
    int array3[10] = {1, 2, 3}; // первые три элемента массива 1,2,3 - остальные ноль

    // инцилазиация многомерных массивов
    int array4[5][5] = {
            {0, 1, 2, 3, 4},
            {0, 1, 2, 3},
            {}
    };

    return 0;
}
```

Массивы нельзя использовать как аргументы функций

Массивы и циклы

```c
#include <stdio.h>
#include <stdint.h>

#define TEMP_ARRAY_SIZE 500

uint8_t getTemperature() {
    uint8_t tempValue;
    // ...
    // операции с датчиком
    // ...
    return tempValue;
}

int main() {
    uint8_t tempValues[TEMP_ARRAY_SIZE];

    for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
        tempValues[i] = getTemperature();
    }

    uint8_t averageTemp = 0;
    for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
        averageTemp += tempValues[i];
    }

    averageTemp /= TEMP_ARRAY_SIZE;
    printf("Среднее значение температуры: %d", averageTemp);
    return 0;
}
```

Циклов в Си несколько: **for**, **while** и **do-while**
Они аналогичны паскалевским for, while и repeat-untill.

for(
то, что происходит перед первой итерацией
;
условие == true
;
то, что происходит после каждой итерации
)
{
Тело цикла
}

Массивы и циклы

```c
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>


bool isLegDeployed() {
    bool status;
    //... работа с датчиками
    return status;
}

void stepLeg(int stepsCount) {
    // ... работа с двигателем ...
}

int main() {

    while(!isLegDeployed()){
        stepLeg(1);
    }

    // do while почти полностью аналогичен
    // но отрабатывает как минимум один раз
    do {
        stepLeg(i);
    } while(isLegDeployed());

    return 0;
}
```

```
while (условие == true) {
   тело цикла
}


do {
    тело цикла
} while(условие == true);
```

Массивы и циклы

```
int main() {

    bool someEventFlag;

    // бесконечный цикл
    while(1) {
        if (someEventFlag){
            break;
        }
    }


    while(1){
        if (someEventFlag)
            continue;

        // действия, которые не произойдут, если continue сработал
    }

    return 0;
}
```

Специальные операции в циклах

break == безусловный выход из цикла
continue == переход к следующей итерации

Массивы и циклы

```c
#include <stdio.h>
#include <stdint.h>

#define TEMP_ARRAY_SIZE 500

uint8_t getTemperature() {
    uint8_t tempValue;
    // ...
    // операции с датчиком
    // ...
    return tempValue;
}

int main() {
    uint8_t tempValues[TEMP_ARRAY_SIZE];

    for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
        tempValues[i] = getTemperature();
    }

    uint8_t averageTemp = 0;
    for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
        averageTemp += tempValues[i];
    }

    averageTemp /= TEMP_ARRAY_SIZE;
    printf("Среднее значение температуры: %d", averageTemp);
    return 0;
}
```

В этой программе есть логическая (не синтаксическая) ошибка. Найдите её

Программа соберется и будет работать, но значение средней температуры будет не верным

Строки и символы

Строковые литералы в двойных кавычках – это на самом деле массивы

```
    // эти определения эквивалентны
    const char string_literal[] = "Какой-то текст";
    const char string_literal2[] = {
          'К', 'а', 'к', 'о', 'й', '-', 'т', 'о',
          ' ',
          'т', 'е', 'к', 'с', 'т',
          0
    };

    printf(string_literal);
    printf(string_literal2);
```

Одиночные символы указываются в одинарных кавычках

Строки и символы

Одиночные символы интерпретируются компилятором как числа

```
#include <limits.h>

int main()
{
    char a = 'a';
    char b = a + 1;
}

char to_upper(char target)
{
    return target + ('A' - 'a');
}
```

Строки и символы

Таблица ASCII

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Обратите внимание, что в char у нас Помещается 255 вариантов символов, а таблица заполнена лишь до 127.

Кодировка ASCII не подразумвает иных символов, но все прочие кодировки, как правило используют ASCII и расширяют её своими символами с номерами от 128 до 255

Стандарт языка си ничего не упоминает о кодировках в которых он принимает символьные литералы, поэтому
Во избежание проблем лучше использовать только английские.

(наш компилятор работает в кодировке utf-8)

Undefined behavior

* Undefined behavior -- behavior, upon use of a non portable or erroneous program construct, for which the standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose.

```
#include <limits.h>

int main()
{
    int the_int;
    int another_int = the_int + 2; // UB!

    signed int sint = INT_MAX - 1;
    sint += 10;     // UB!

    unsigned int uint = UINT_MAX;
    uint += 1;      // вот так можно

    int thearray[10];
    thearray[50] = 10;      // UB!
    theint = thearray[10];  // UB!
}
```