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, написаных 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 компиляторы семейства **GCC** как для настольного компьютера так и для микроконтроллера.

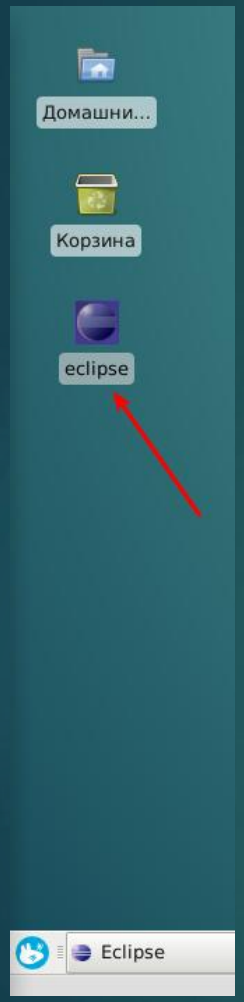
Компилятор, стандартная библиотека и среда разработки

- 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.

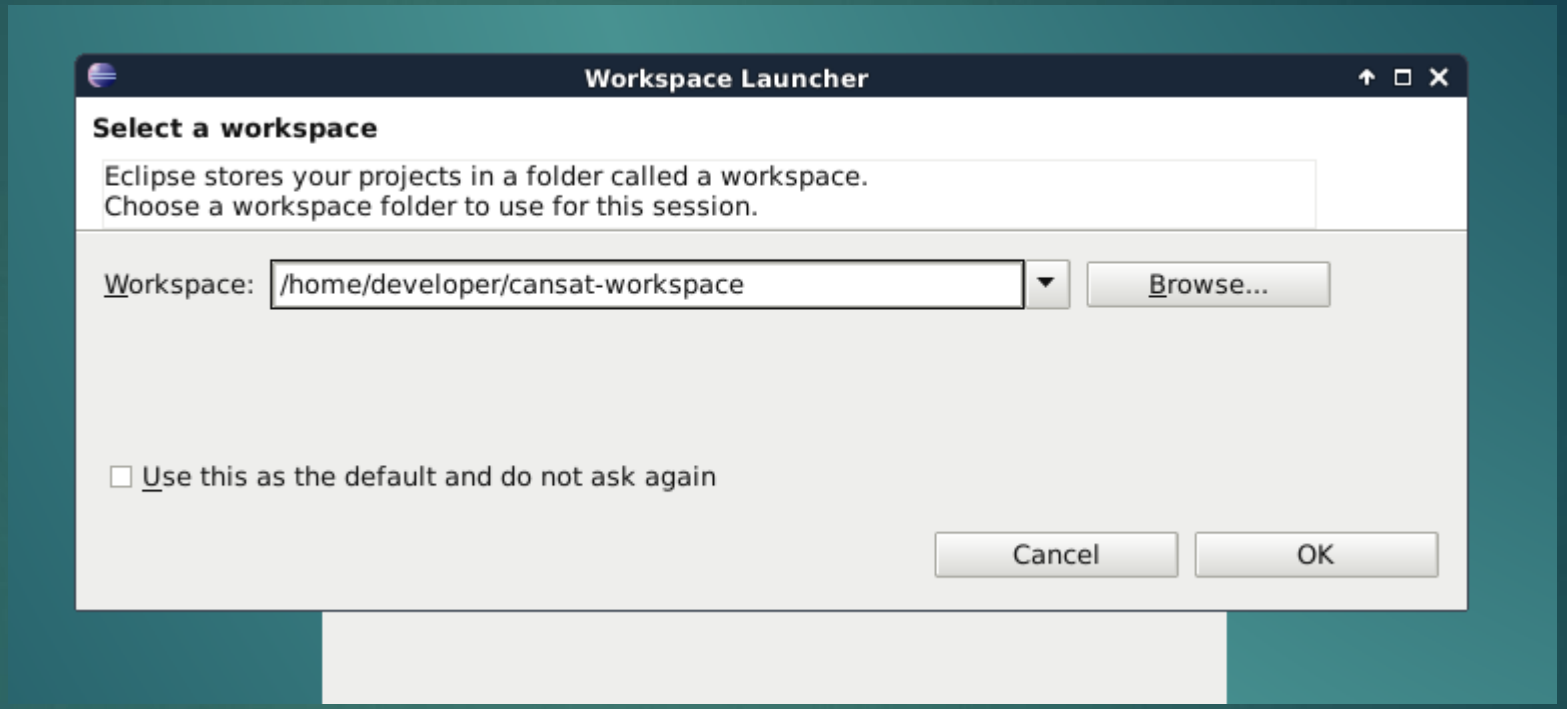
Компилятор, стандартная библиотека и среда разработки

- 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!

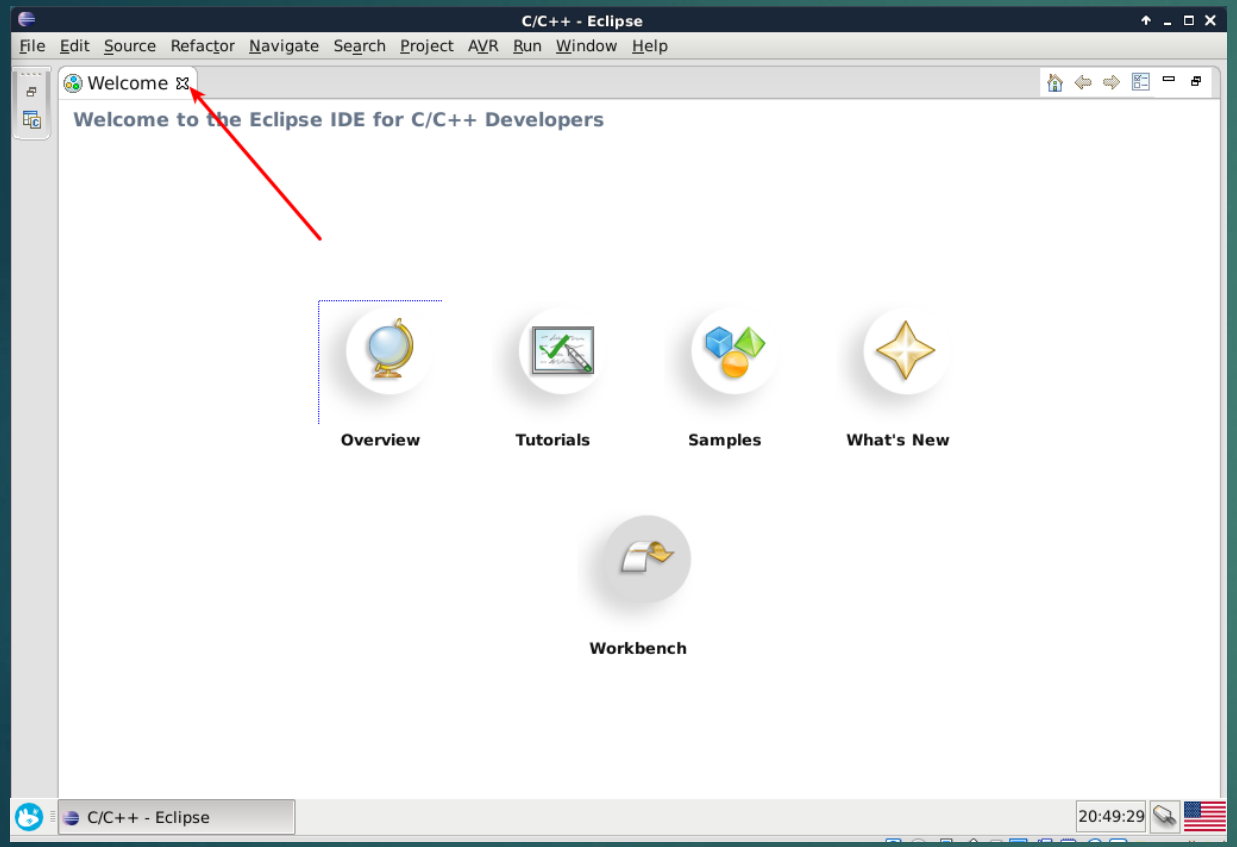


1. Запускаем eclipse

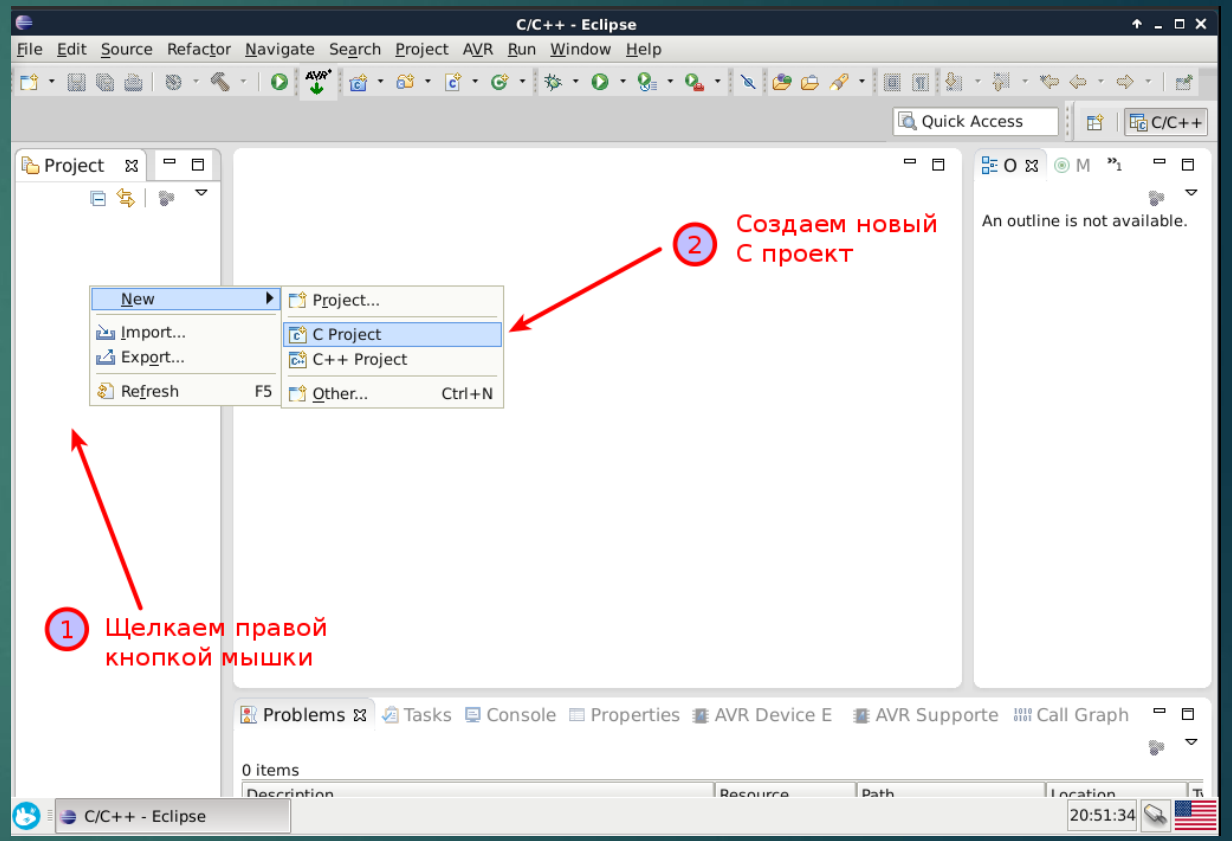


2. Создаем/выбираем workspace

Программа «Hello world»

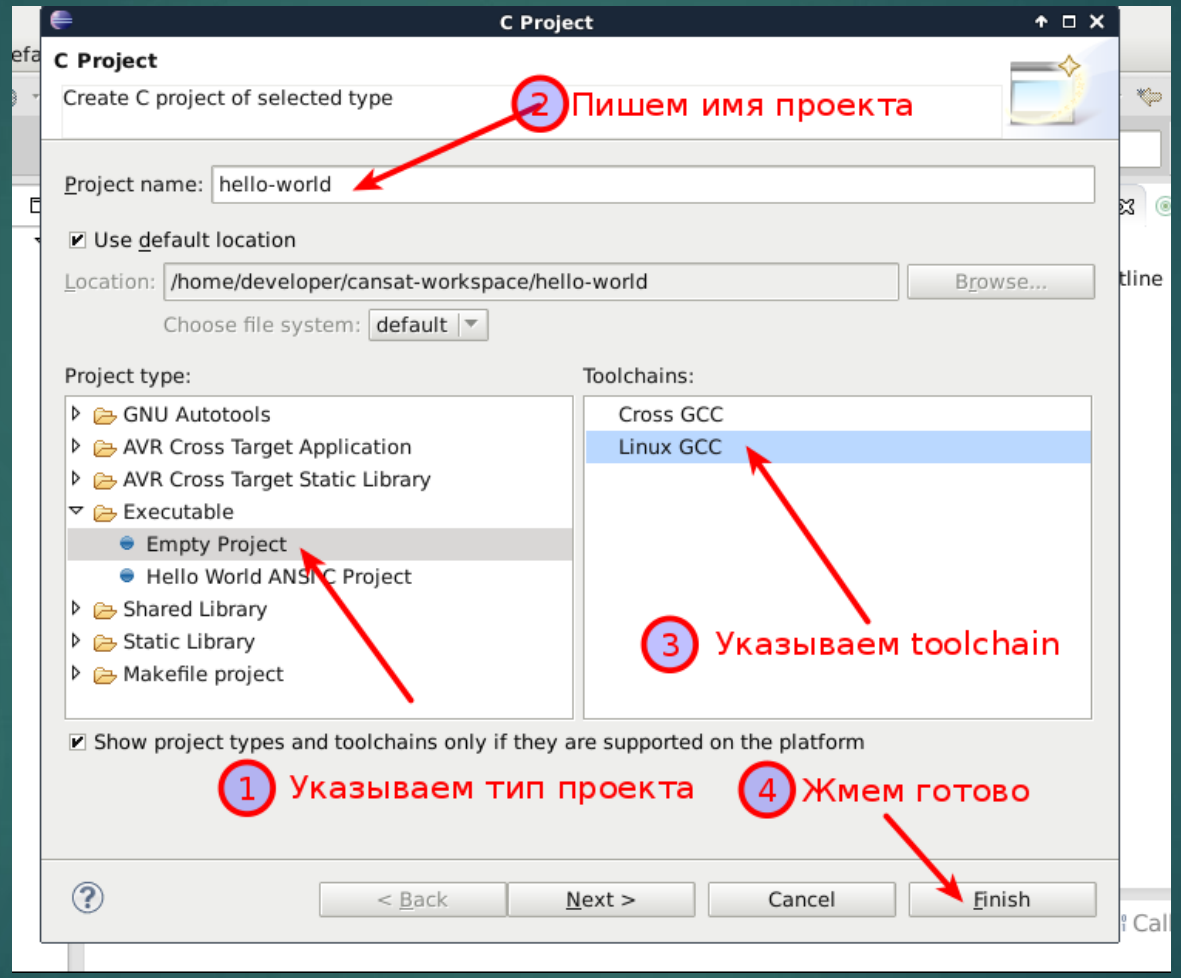


2. Закрываем страницу приветствия



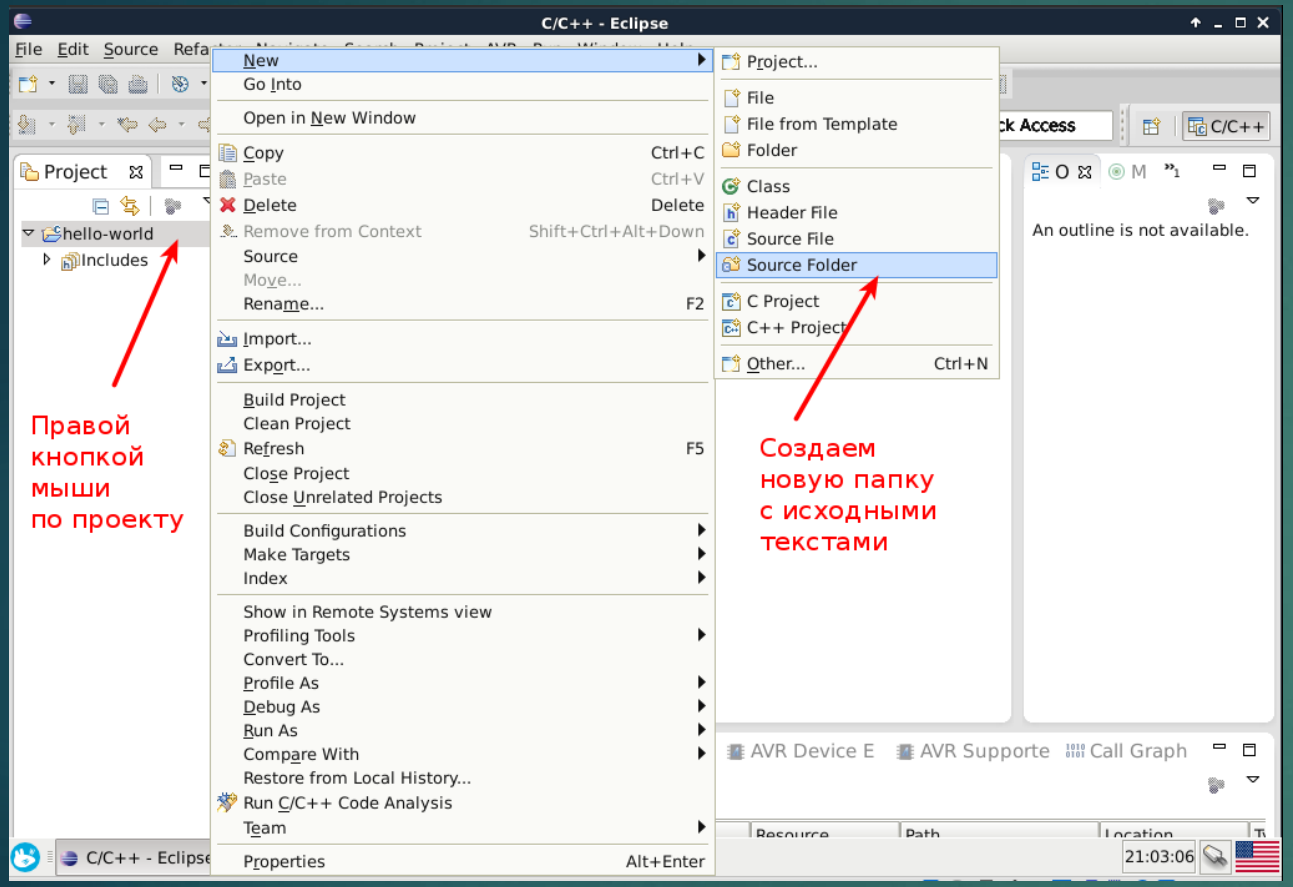
3. Создаем новый С проект

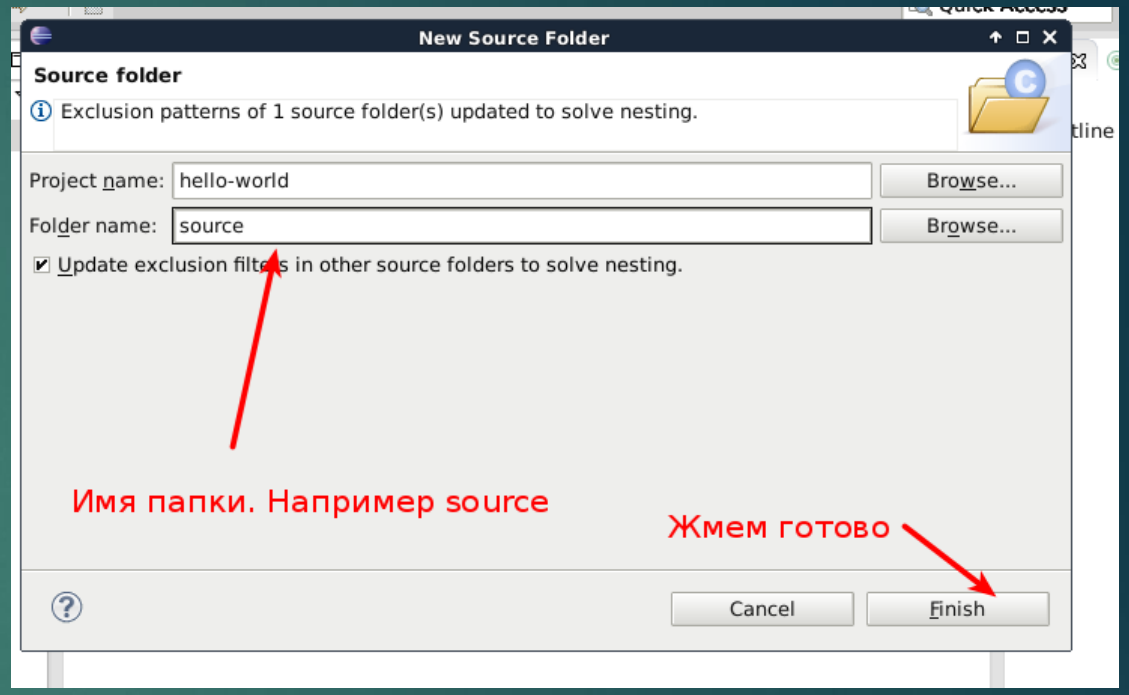
Программа «Hello world»



4. Указываем параметры проекта

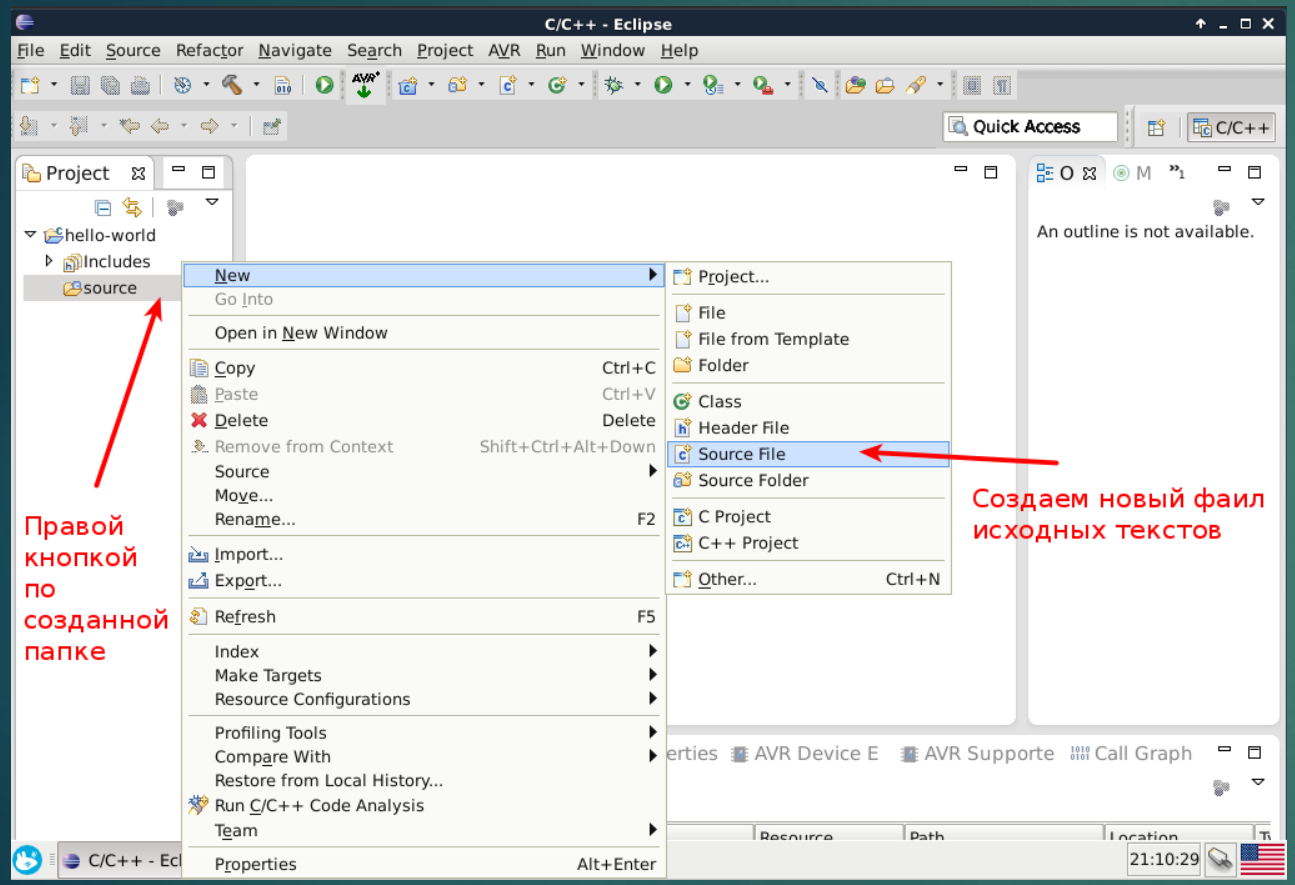
Программа «Hello world»

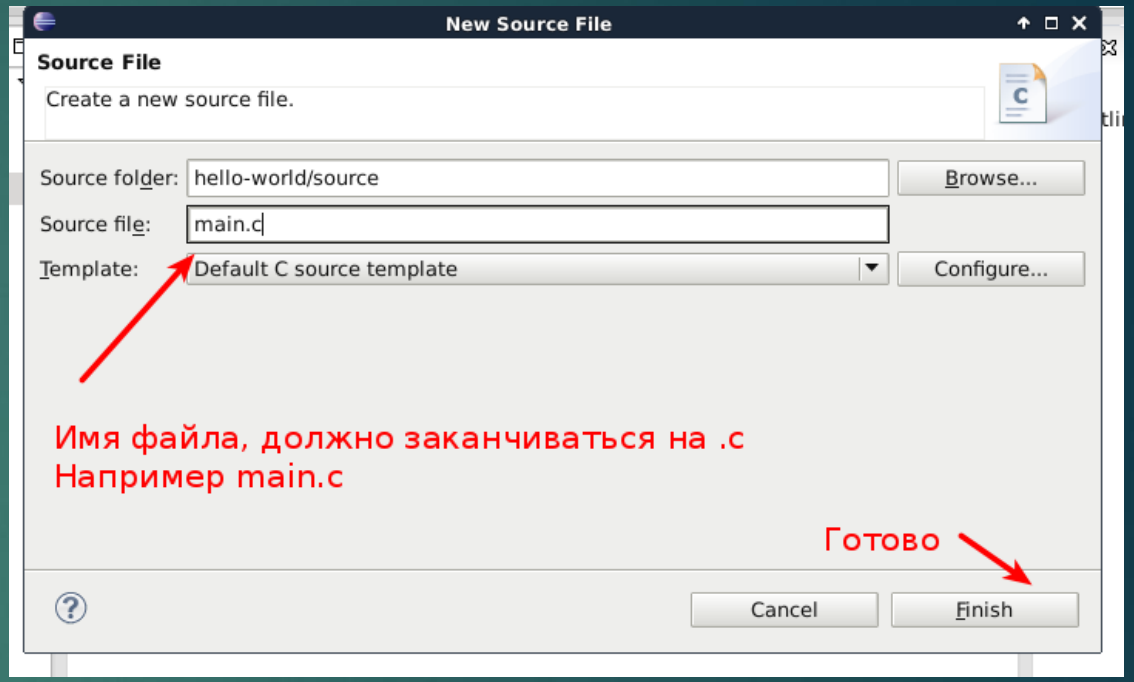




5. Создаем папку для исходных текстов проекта

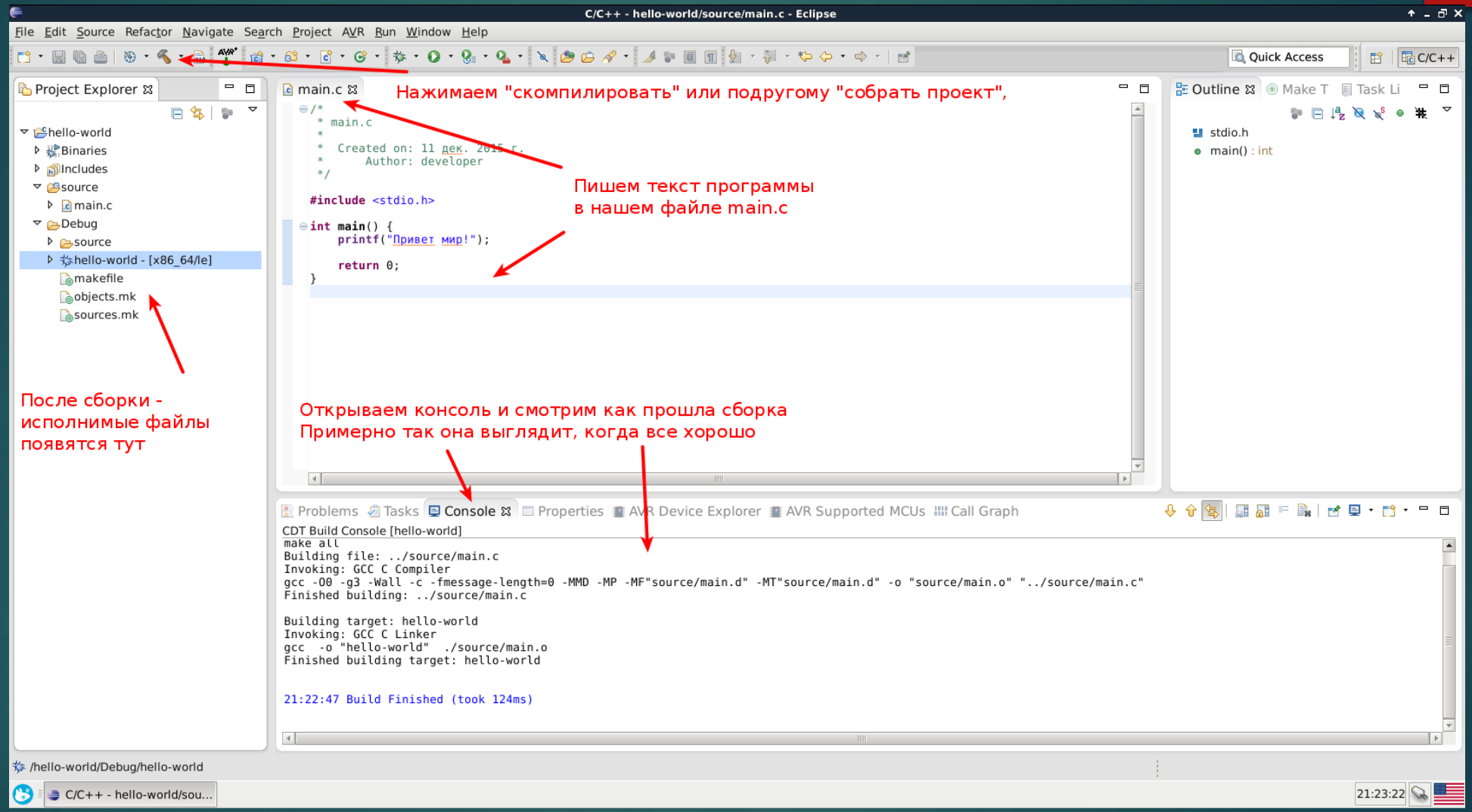
Программа «Hello world»





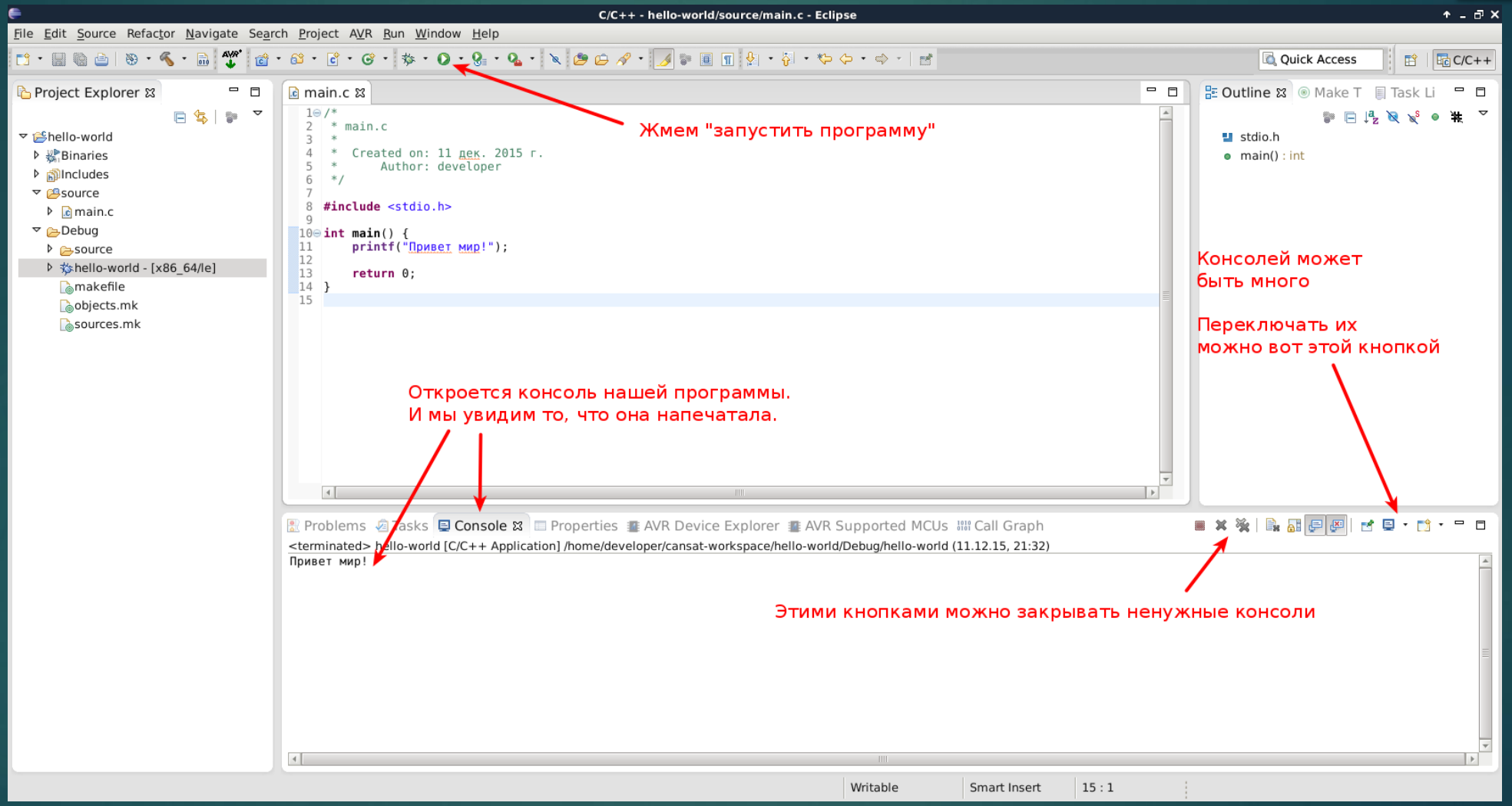
5. создаем файл исходных текстов

Программа «Hello world»



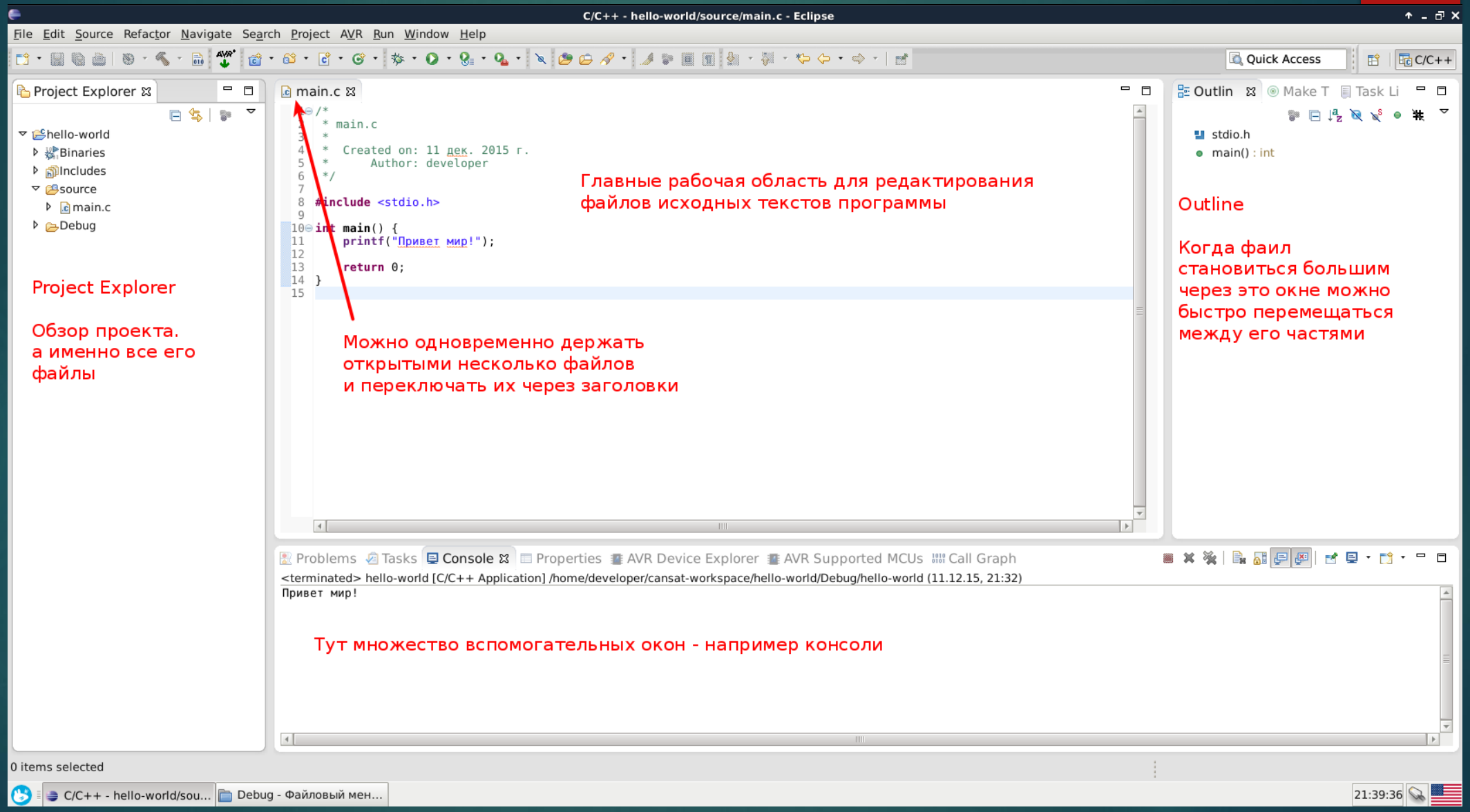
6. Собираем программу

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»

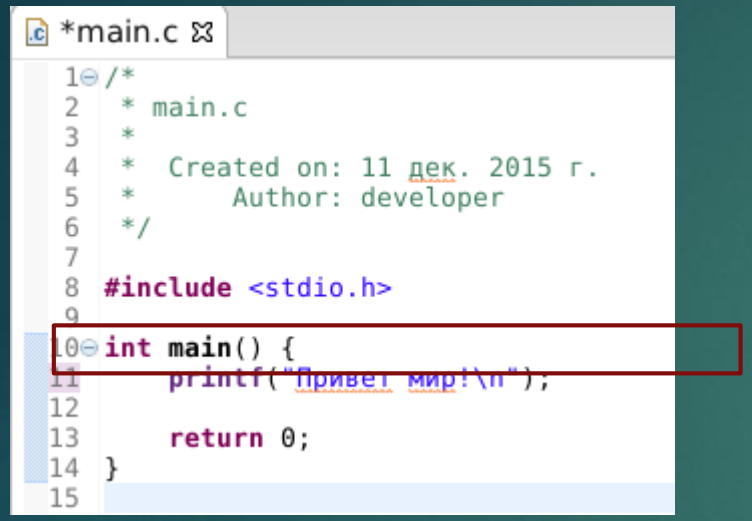
#include – это директива по которой препроцессор – берёт фаил с именем **stdio.h** из стандартных каталогов и включает его содержимое в наш main.c как будто бы мы скопировали весь текст из него и вставили бы вместо директивы

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

```
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
```

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.

```
8  #include <stdio.h>
9
10
11 int main() {
12     printf("Привет мир!\n");
13
14     return 0;
15     printf("Эта операция не выполнится");
16 }
17
```

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

Помимо указания вовзращаемого значения – return завершает функцию и все, что идет после него – не будет выполнено.

Подробнее о функциях

- Определение функции – задание кода функции. Должно быть сделано только один раз для каждой функции во всех исходных текстах программы. Например:

```
12
13 int summ(int arg1, int arg2) {
14     return arg1 + arg2;
15 }
16
```

- Функция не может быть определена внутри другой функции

```
13 int outerFunction(int arg1, int arg2, int arg3) {
14
15     // так делать нельзя
16     int innerFunction(int arg11, int arg12) {
17         return arg11 + arg12;
18     }
19
20     return innerFunction(arg1, arg2) + arg3;
21 }
```

Вызов функции

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

- Например:

```
 8 #include <stdio.h>
 9
10 void print_number(int number) {
11     printf("Привет %d\n", number);
12 }
13
14 int main() {
15     print_number(10);
16     print_number(20);
17     print_number(30);
18
19     return 0;
20 }
```

Вывод:

Привет 10

Привет 20

Привет 30

следует отметить, что параметры переданные функции копируются и она работает со своими личными копиями

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.

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]

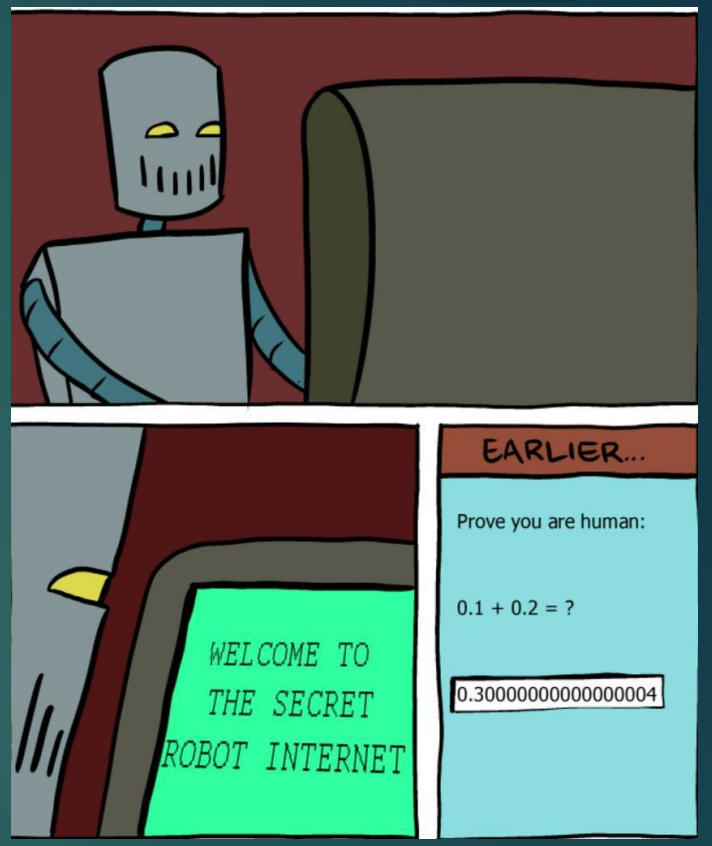
Так же может быть полезен файл **limits.h** (#include <limits.h>) в нем определены минимальные/максимальные значения целочисленных типов в макросах **SHRT_MIN, SHRT_MAX, USHRT_MIN, USHRT_MAX** и прочие.

Типы с плавающей точкой

Их всего два

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

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

Сравнение

```
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;
```

Ветвление программы

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

- Например:

```
 8 #include <stdio.h>
 9
10 int main() {
11     int x = 10;
12
13     if (x > 5)
14     {
15         printf("X больше пяти");
16     }
17     else
18     {
19         printf("X не больше пяти");
20     }
21
22     return 0;
23 }
```