

Конспект лекций Мещерина

Содержание

1 Введение	1
1.1 Общие слова о языке и исторические заметки	1
1.2 Знакомство с компилятором и первая программа . . .	3
1.3 Виды ошибок и неопределенное поведение (UB) . . .	4
1.4 Declarations, definitions and scopes.	18
1.5 Basic types and supported operation.	24

1. Введение

Лекция 1

О лекторе

Лектора зовут Мещерин Илья Семирович.

VK: vk.com/mesyarik

TG: @mesyarik

1.1. Общие слова о языке и исторические заметки

Почему изучается именно этот язык программирования и что он даст? На сайте www.tiobe.com в котором собрана информация о популярности языков программирования C++ занимает достаточно высокое место.

Существуют языки **общего** и **специального** назначений, например на C++, C#, Python, Java можно писать почти все что угодно, а на JavaScript — нет.

По этому курсу будет довольно много лекций, потому что C++ довольно большой язык. Вместе с изучением C++ будет затронут и его прородитель — C.

Почти всегда то, что написано на C++ будет работать и на C. Также, изучая C++ будет намного проще изучать Java и C#, потому что эти языки синтаксически похожи.

Язык C++ — очень сложен, освоив его изучать остальное будет проще. Сложность освоения C++ связана со многими вещами, как минимум он довольно низкоуровневый. Придется следить за памятью, есть возможность общаться с ОС на прямую и т.д.

В Java и C# мы защищены от низкоуровневых инструкций, что делают их проще и безопаснее. Однако в этом есть и плюсы и минусы. Из-за незащищенности и низкоуровневости код C++ может работать в разы быстрее своих аналогов. Исходя из этого C++ используется там, где нужен высокопроизводительный код.

Примеров известных сервисов, которые написаны с использованием кода на C++ довольно много.

Яндекс: Поиск, Карты, Такси, Почта, Диск.

Google: Search, Youtube.

Телеграмм.

Также на C++ написано множество игр, например WoW.

Не стоит и забывать про ОС, даже Windows написан с использованием C++.

C++ используется и в науке.

C++ создал Бьерн Страуструп (Bjarne Stroustrup). В курсе его будем называть **создателем**.

У C++ много версий. Мы будем пользоваться стандартом C++ 17.

- C++98
- C++03

- C++11 (C++0x)
- C++14 (C++1y)
- C++17 (C++1z)

Между версиями языка не зря проведена черта. Они очень сильно отличаются. У каждой версии есть **стандарт**. Тонны страниц текста с мелким шрифтом, исчерпывающе описывающим версию языка. Из-за его объема и формальности не рекомендуется читать и вникать в него полностью, а лишь обращаться к нему и читать по диагонали в неоднозначных моментах.

Рекомендуется пользоваться сайтом www.cppreference.com. Если требуется что-то узнать о языке, то первым делом лучше лезть туда.

Не стоит брезговать сайтом www.stackoverflow.com, на котором почти всегда будут ответы на возникающие вопросы.

Рекомендованная литература, автора Скота Мейерса:

- Эффективное использование C++.
- Наиболее эффективное использование C++.
- Эффективный и современный C++.

1.2. Знакомство с компилятором и первая программа

Добро пожаловать в терминал Linux! В курсе рекомендуется пользоваться именно этой операционной системой.

Напишем первую программу.

```
#include <iostream>

int main() {
    int x;
    std::cin >> x;
    std::cout << x + 5 << '\n';
}
```

В программе на C++ обязательно должна быть такая функция как **main**. Здесь мы пишем программу от которой ожидаем простые действия, а именно ввести число, увеличить его на пять и вывести.

Чтобы делать ввод-вывод необходимо подключить **заголовочный файл** — [iostream](#).

Написав строку

```
#include <iostream>
```

мы подключаем заголовочный файл и получаем возможность работать со стандартными потоками ввода-вывода.

Для того чтобы ввести переменную, необходимо ее **объявить**.

```
int x;
```

Чтобы ввести переменную из стандартного потока ввода пишем

```
std::cin >> x;
```

а чтобы вывести число в стандартный поток вывода из этой переменной увеличенное на 5 пишем

```
std::cout << x + 5 << '\n';
```

Здесь мы также добавляем к выводу **символ перевода строки** — '\n'.

Чтобы перевести строку можно написать и

```
std::cout << x + 5 << std::endl;
```

однако есть разница. Во втором случае мы также отчищаем буффер вывода.

Файлы исходного кода обычно выглядят как *.cpp.

Перейдем к компиляции. Для того чтобы получить исполняемую программу необходимо перевести исходный код в машинный, этот процесс называется **компиляцией**, а наличие этого процесса отличает компилируемые языки программирования от интерпретируемых.

Для того чтобы скомпилировать исходный код необходимо вызвать специальную программу — **компилятор**. У C++ есть несколько компиляторов. Наиболее известным из них является gcc — Gnu Compiler Collection. Для того чтобы его вызвать именно для языка C++ необходимо писать g++.

Также есть такие компиляторы, как clang++ и msvc. Компилятор msvc разрабатывается компанией Microsoft и не рекомендован к использованию, поскольку многие вещи от туда не всегда соответствуют стандарту.

Для отладки программы можно пользоваться программой дебагером, например gdb.

Лекция 2

1.3. Виды ошибок и неопределенное поведение (UB)

C++ — компилируемый язык. Поговорим о том, какие ошибки бывают ошибки при компиляции и после нее.

Программа бывает некорректна по многим причинам. Рассмотрим ошибку компиляции **CE** Compilation Error.

Ошибки компиляции бывают разные. Например, если где-то забыть поставить точку с запятой, то это будет **синтаксическая ошибка**. Она случается когда компилятор не смог распарсить то что написано из-за синтаксиса. Ошибкой компиляции также считается использование переменной или функции, которые не были объявлены.

При написании

```
#include <iostream>
```

Мы вставляем в исходный код содержание заголовочного файла, в данном случае `iostream`, в котором содержатся объявления функций `std::cin` и `std::cout`.

И если забыть подключить заголовочный файл и воспользоваться функцией, то это будет СЕ.

Ошибкой будет и повторное объявление переменной в одной и той же области видимости.

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

Поэтому подобный код тоже будет выдавать ошибку СЕ при попытке компиляции.

Также при обычных обстоятельствах недопустимо пользоваться переменной/функцией до объявления.

```
#include <iostream>
```

```
int main() {  
    std::cout << x << '\n';  
    int x;  
}
```

Выдаст СЕ при попытке компиляции.

В целом ошибки компиляции можно разбить на три группы:

- Лексические ошибки
- Синтаксические ошибки
- Семантические ошибки

Компиляция это очень сложный процесс, который состоит из разных стадий, поговорим о трех. Так, в начале происходит лексический разбор, потом синтаксический, а потом компилятор разбирается в семантике того что написано.

Разберем поподробнее. При лексическом разборе компилятор пытается разбить код на осмысленные последовательности символов — **токены**. Пример:

```
std::cout << x;
```

Этот код будет разбиваться на токены следующим образом:

`(std)(::)(cout)(<<)(x)(;)(.).`

Здесь круглыми скобками обозначены токены, на которые разберется код при лексическом разборе.

Возникает вопрос, почему компилятор определил («) а не например как два знака меньше (<)(<)? На самом деле действует следующее правило. Компилятор будет идти слева направо и пока это осмысленно он будет добавлять символы к токену. Именно поэтому не получится разбиения (std:), поскольку это неосмысленно, а вот (std) — осмысленно.

Если компилятору не удалось разбить код на токены на этапе лексического разбора, то такая ошибка будет называться **лексической**.

```
#include <iostream>

int main() {
    \\\\;

    int x;
    std::cin >> x;
    std::cout << x + 5;
}
```

При компиляции кода можно увидеть следующее, рисунок 1.

Здесь stray можно перевести как "заблудший" или "заплутавший". Компилятор не смог это как-то интерпретировать.

Приведем более подробные примеры синтаксических ошибок.

```
#include <iostream>

int main() {
    int x
    std::cin >> x;
    std::cout << x + 5;
}
```

На рисунке 2 написано, что компилятор ожидал инициализацию перед std. Иначе говоря он парсит строки 5 и 6 как одно выражение, так как в строке пять нет ';', однако после объявления переменной компилятор ожидает либо ничего, либо ее инициализацию, а так как дальнейшее инициализацией никак не является, то это ошибка. Так как эта ошибка возникла из-за отсутствия ';' и была обнаружена на этапе синтаксического разбора она считается синтаксической.

```
g++ --std=c++17 l2/e1.cpp -o program
l2/e1.cpp:5:5: error: stray '\ ' in program
  5 |     \\\;
    |     ^
l2/e1.cpp:5:6: error: stray '\ ' in program
  5 |     \\\;
    |     ^
l2/e1.cpp:5:7: error: stray '\ ' in program
  5 |     \\\;
    |     ^
l2/e1.cpp:5:8: error: stray '\ ' in program
  5 |     \\\;
    |     ^
l2/e1.cpp:5:9: error: stray '\ ' in program
  5 |     \\\;
```

Рис. 1: Пример лексической ошибки.

Далее компиляция не прервалась и было обнаружено, что переменная `x` не была объявлена, поскольку предыдущее выражение ошибочно. Стоит отметить, что компилятор не сказал об использовании необъявленной переменной в 5-ой строке после `std::cin`, так как уже отловил в этом выражении синтаксическую ошибку. Выражения не всегда рассматриваются на предмет ошибок до конца, да и здесь это не требуется.

```
#include <iostream>

int main() {
    int x;
    std::cin >> x;
    std::cout << x + 5;
    x + 5 + ;
}
```

Поскольку `+` здесь используется как бинарный оператор, а после стоит `;`, то перед `;` ожидается некое primary-expression, о котором будет рассказано позже.

Компилятор старается не завершать компиляцию, чтобы выдать как можно больше ошибок сразу, однако у него не всегда это

```

g++ -std=c++17 l2/e2.cpp -o program
l2/e2.cpp: In function 'int main()':
l2/e2.cpp:5:5: error: expected initializer before 'std'
   5 |     std::cin >> x;
     |     ^~~
l2/e2.cpp:6:18: error: 'x' was not declared in this scope
   6 |     std::cout << x + 5;
     |                  ^

```

Рис. 2: Пример синтаксической ошибки 1.

```

g++ -std=c++17 l2/e3.cpp -o program
l2/e3.cpp: In function 'int main()':
l2/e3.cpp:7:13: error: expected primary-expression before ';' token
   7 |     x + 5 + ;
     |              ^

```

Рис. 3: Пример синтаксической ошибки 2.

получается. Ошибки, при которых компиляция останавливается, называются **фатальными**.

```

#include <iostream>
#include <aaaaaaaa>

int main() {

}

```

```

g++ -std=c++17 l2/e4.cpp -o program
l2/e4.cpp:2:10: fatal error: aaaaaaaaa: No such file or directory
   2 | #include <aaaaaaaa>
     |          ^~~~~~
compilation terminated.

```

Рис. 4: Пример синтаксической ошибки 3. Фатальная ошибка.

Примером фатальной ошибки будет являться подключение файла, которого не существует.

Можно привести такой неформальный пример синтаксической ошибки

Я пошел лес грибы.

Вроде и все понятно, но синтаксически неверно.

А вот какой пример можно привести для семантической ошибки

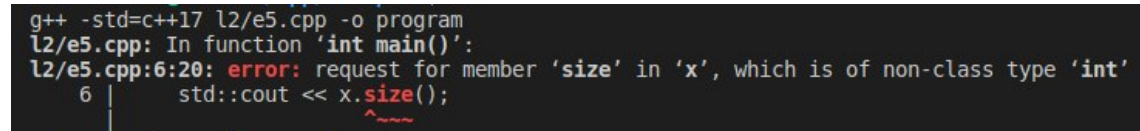
Я ем стол.

С точки зрения синтаксиса здесь все верно, однако есть стол это как-то странно. Иначе говоря мы понимаем, какую пищу может есть человек, а какую нет. Человек не может есть стол. Подобная ошибка называется семантической.

Другими словами если мы пытаемся совершить что-то невозможное, то это **семантическая ошибка**.

```
#include <iostream>
```

```
int main() {  
    int x;  
    std::cin >> x;  
    std::cout << x.size();  
}
```

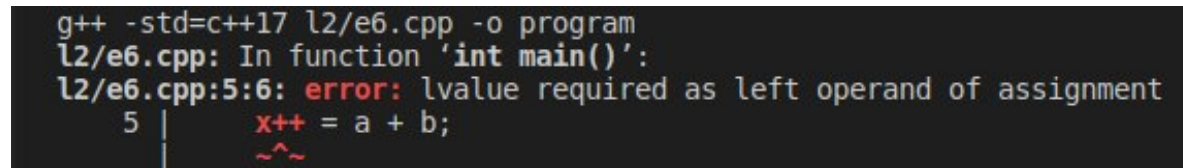


```
g++ -std=c++17 l2/e5.cpp -o program  
l2/e5.cpp: In function 'int main()':  
l2/e5.cpp:6:20: error: request for member 'size' in 'x', which is of non-class type 'int'  
6 |     std::cout << x.size();  
  |                   ^~~~~
```

Рис. 5: Пример семантической ошибки 1.

Мы пытаемся применить метод `.size()` к переменной целочисленного типа, однако это невозможно, поскольку переменная целочисленного типа не обладает методом `.size()`. О методах мы узнаем позже.

```
int main() {  
    int x;  
    int a;  
    int b;  
    x++ = a + b;  
}
```



```
g++ -std=c++17 l2/e6.cpp -o program  
l2/e6.cpp: In function 'int main()':  
l2/e6.cpp:5:6: error: lvalue required as left operand of assignment  
5 |     x++ = a + b;  
  |     ~^~
```

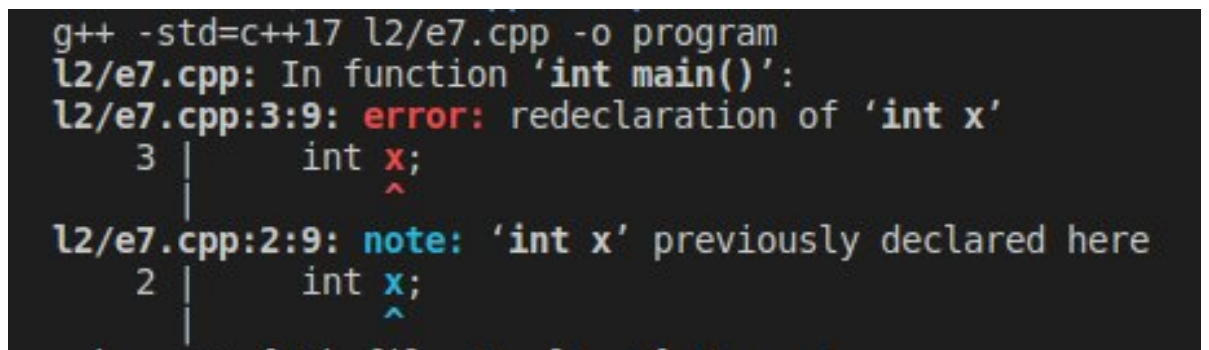
Рис. 6: Пример семантической ошибки 2.

Это тоже семантическая ошибка, поскольку тому, что стоит слева от '=' невозможно присвоить что-либо, так как то что слева не

является lvalue. О том что такое lvalue мы тоже поговорим позднее.

Заметим, что на рисунке 2 также показана семантическая ошибка, так как мы пытаемся использовать необъявленную переменную.

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



```
g++ -std=c++17 l2/e7.cpp -o program  
l2/e7.cpp: In function 'int main()':  
l2/e7.cpp:3:9: error: redeclaration of 'int x'  
  3 |     int x;  
    |     ^  
l2/e7.cpp:2:9: note: 'int x' previously declared here  
  2 |     int x;  
    |     ^
```

Рис. 7: Пример семантической ошибки 3.

Вспомним о повторном объявлении переменной. Это тоже семантическая ошибка, рисунок 7.

Также иногда семантическая ошибка возникает при неоднозначном обращении, рисунок 8.

```
#include <iostream>  
  
namespace N {  
    int x;  
}  
  
using namespace N;  
  
int x = 0;  
  
int main() {  
    std::cin >> x;  
    std::cout << x + 5;  
}
```

Здесь переменная 'x' объявлена в пространстве имен 'N', а далее, строкой

```
using namespace N;
```

```

g++ -std=c++17 l2/e8.cpp -o program
l2/e8.cpp: In function 'int main()':
l2/e8.cpp:12:17: error: reference to 'x' is ambiguous
   12 |     std::cin >> x;
       |                  ^
l2/e8.cpp:4:9: note: candidates are: 'int N::x'
   04 |     int x;
       |         ^
l2/e8.cpp:9:5: note:                  'int x'
   09 | int x = 0;
       |     ^
l2/e8.cpp:13:18: error: reference to 'x' is ambiguous
   13 |     std::cout << x + 5;
       |                  ^
l2/e8.cpp:4:9: note: candidates are: 'int N::x'
   04 |     int x;
       |         ^
l2/e8.cpp:9:5: note:                  'int x'
   09 | int x = 0;
       |     ^

```

Рис. 8: Пример семантической ошибки 4.

сливается с глобальной областью видимости.

Так как переменные объявлены в разных областях видимости здесь нет ошибки повторного объявления, значит это две разные переменные. Однако после сливания не понятно, к какой именно переменной идет обращение.

Помимо ошибок компиляции существуют ошибки, которые возникают во время исполнения программы. Они называются **Runtime error (RE)**.

Примеры ошибок времени выполнения:

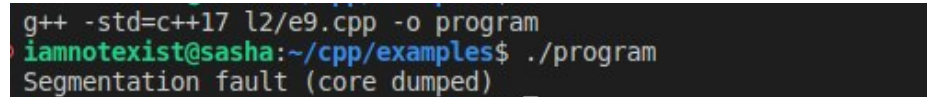
1. Слишком далекий выход за границу массива
2. Бесконечная рекурсия
3. Целочисленное деление на ноль

Первые и вторые примеры это примеры одной ошибки, которая называется **Segmentation fault**. При этой ошибке программа обращается к памяти, которая ей не принадлежит.

```
#include <iostream>
```

```
int main() {
    int a[100];

    a[100000000] = 1;
}
```



```
g++ -std=c++17 l2/e9.cpp -o program
iamnotexist@sasha:~/cpp/examples$ ./program
Segmentation fault (core dumped)
```

Рис. 9: Segmentation fault.

При попытке исполнения этой программы получим результат Segmentation fault. Так как программа работает с памятью, которую ей выдала операционная система, работа с не ее памятью может привести к прекращению работы. Таким образом борется с тем что мы пытаемся работать не со своей памятью.

Аналогичный результат будет и при бесконечной рекурсии:

```
void f() {
    f();
}

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

Здесь Segmentation fault происходит потому, что при вызове функции программе необходимо запомнить адрес возврата. Поскольку вызовы не будут прекращаться, программа привысит лимит выделенной памяти и попытается записать информацию в память, которая ей не принадлежит. Так как адреса возврата запоминаются на стеке, то это приводит к его переполнению — **Stack overflow**.

Третий же пример ошибки выполнения под названием **Floating point exception**.

```
#include <iostream>

int x;

int main() {
    std::cin >> x;
    std::cout << x / 0;
}
```

```

g++ -std=c++17 l2/e11.cpp -o program
l2/e11.cpp: In function 'int main()':
l2/e11.cpp:7:20: warning: division by zero [-Wdiv-by-zero]
    7 |         std::cout << x / 0;
      |         ~~~~~^~
iamnotexist@sasha:~/cpp/examples$ ./program
4
Floating point exception (core dumped)

```

Рис. 10: Floating point exception.

При компиляции компилятор выдал предупреждение **Warning**, информацию о том, что в коде возможно что-то не так. При попытке запуска и после ввода значения программа падает с ошибкой Floating point exception.

По другому падение программы при подобных ошибках называют **аварийным завершением программы**. Также примером RE бывает **Illegal instruction**. Это происходит когда машинный код поврежден и процессор не может его выполнить.

Помимо этого в C++ есть и другие варианты ошибок. Поговорим про **Undefined behavior**, что переводится как неопределенное поведение. Неопределенное поведение это когда при некоторой ситуации в коде программы компилятор перестает гарантировать что либо относительно поведения такого кода.

Приведем абстрактный пример. Пусть есть олимпиадная задача по информатике, в который подробно описаны входные данные. Решением будет программа, которая обработает входные данные исходя из ограничений и выдаст ответ. Но что она будет делать если ей гарантировали, к примеру, неотрицательные числа во входных данных, а дают отрицательные? По сути программа может делать все что угодно.

Так и с компилятором. У компилятора есть входные данные — код на C++, которые ограничены его стандартом. И в самом стандарте прописаны некоторые оговорки о том, что при каких-то действиях будет возникать UB.

При наличии в коде инструкций, которые классифицируются как UB, компилятор может делать все что угодно. Существует даже шутка о том, что компилятор может поджечь монитор. Тем самым при наличии в коде UB код является некорректными данными для компилятора.

Примеры неопределенного поведения:

1. Выход за границу массива.

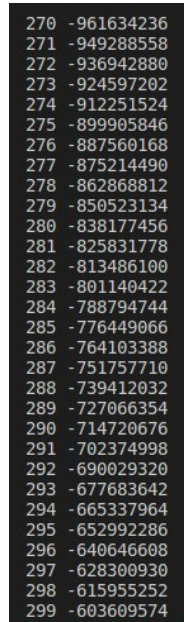
2. Переполнение переменной типа `int`.

При выходе за границу массива возможен RE, но он также может и не произойти. При этом RE будет следствием UB.

Приведем пример UB при переполнении `int`.

```
#include <iostream>
int main() {
    for (int i = 0; i < 300; i++)
        std::cout << i << " " << i * 12345678 << std::endl;
}
```

Здесь будет выводиться 'i' и 'i умноженное на 12345678', после будет перевод строки. Все это будет происходить в цикле. Попробуем скомпилировать и запустить.



```
270 -961634236
271 -949288558
272 -936942880
273 -924597202
274 -912251524
275 -899905846
276 -887560168
277 -875214490
278 -862868812
279 -850523134
280 -838177456
281 -825831778
282 -813486100
283 -801140422
284 -788794744
285 -776449066
286 -764103388
287 -751757710
288 -739412032
289 -727066354
290 -714720676
291 -702374998
292 -690029320
293 -677683642
294 -665337964
295 -652992286
296 -640646608
297 -628300930
298 -615955252
299 -603609574
```

Рис. 11: Вывод примера 12.

На выходе получаем рисунок 11.

В начале вывода числа будут нормальными, но в потом они будут отрицательными. На 174 шаге `int` переполнился.

Поробуем скомпилировать код с параметром оптимизации **-O2**. Параметр оптимизации говорит компилятору о том, что если код можно преобразовать в тождественно такой же код, но более

простой, то он это делает. В зависимости от параметра он может делать это по-разному.

Бывают следующие параметры

- -O0. При этом параметры оптимизации не производятся.
- -O1. Простые оптимизации.
- -O2. Более мощные оптимизации.
- -O3. Максимальные оптимизации.

При флаге -O2 код будет работать быстрее.

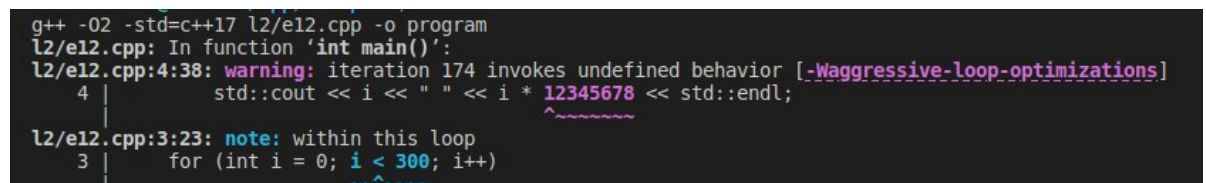


Рис. 12: Вывод примера 12 при компиляции с флагом -O2.

Компилятор заметил, что на 174 итерации цикла происходит UB. При попытке запуска программы она входит в вечный цикл или **зацикливается**.

После оптимизации компилятором цикл, который шел 300 итераций, вдруг заиклился. Компилятор увидел UB из-за aggressive-loop-optimization. Это случилось потому, что компилятор заметил, что 'i' умножается на '12345678'. В предположении того, что код корректен, 'i' не может превзойти 173, так как в предположении корректности int не переполняется. Значит условие в цикле 'i < 300' никогда не будет ложным, а значит, для избежания лишних сравнений компилятор может подставить туда истину. Поэтому программа и заикливается.

UB — самое плохое, что может быть в коде. Дело в том, что например ошибки компиляции идут во благо. То, к чему компилятор придирается, дает возможность избежать ошибок в run-time. Если компилятор чего-то не нашел, значит может произойти что-то плохое в run-time, что будет сложно отловить. Существуют даже специальные слова, которые не влияют на код, но без них компиляция не будет происходить.

Почему тогда компилятор не отлавливает RE во время компиляции? Просто потому что это может быть невозможно с точки зрения математики. У этого даже существует название — **проблема остановки**. Иначе говоря, глядя на программу, невозможно всегда сказать, произойдет ли какое-то действие или нет.

Почему UB нельзя классифицировать как RE и аварийно завершать программу? Например в Java так и происходит, однако C++ не про это. C++ нацелен на производительность и эффективность.

Абстрактный пример:

Если мама постоянно будет проверять, надели ли мы шапку, перед тем как пойти на улицу, это в скором времени начнет нас бесить.

Проверки в run-time требуют определенного времени, а мы не хотим терять в производительности. За счет существования UB C++ выигрывает в эффективности.

Помимо UB в C++ существует неспецифицированное поведение **Unspecified behavior (UC)**. Иначе говоря, в некоторых ситуациях у компилятора есть ограниченная свобода действий. Однако результат должен быть в каких-то рамках.

```
#include <iostream>
```

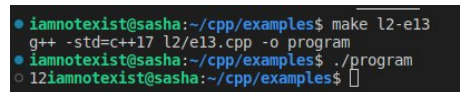
```
int f() {  
    std::cout << 1;  
    return 0;  
}
```

```
int g() {  
    std::cout << 2;  
    return 0;  
}
```

```
int main() {  
    f() + g();  
}
```

Вывод этой программы будет неспецифицированным. Мы не можем с уверенностью сказать, что выведет эта программа. По стандарту она может выводить как 12 так и 21. Это связано с тем, что компилятор может запускать функции в данном случае в любом порядке.

Поробуем скомпилировать и запустить эту программу.

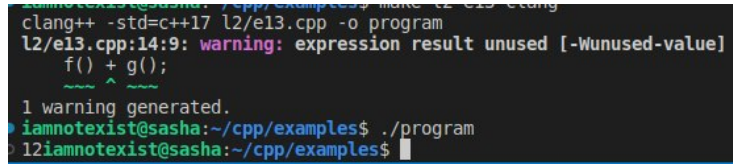


```
iamnotexist@sasha:~/cpp/examples$ make l2-e13  
g++ -std=c++17 l2/e13.cpp -o program  
iamnotexist@sasha:~/cpp/examples$ ./program  
12iamnotexist@sasha:~/cpp/examples$
```

Рис. 13: Пример неспецифицированного поведения, g++.

В данном случае программа вывела 1 потом 2.

Попробуем скомпилировать другим компилятором и запустить.



```
iamnotexist@sasha:~/cpp/examples$ clang++ -std=c++17 l2/e13.cpp -o program
l2/e13.cpp:14:9: warning: expression result unused [-Wunused-value]
    f() + g();
    ~~~ ^ ~~~
1 warning generated.
iamnotexist@sasha:~/cpp/examples$ ./program
12iamnotexist@sasha:~/cpp/examples$
```

Рис. 14: Пример неспецифицированного поведения, clang++.

Вывод программы остался тем же, однако заметим разницу в компиляторах. clang++ дал информацию о том, что мы не используем результат вычисления функций, выдавая Warning.

В стандарте нет требования о том, что при вычислении выражения 'a + b' компилятор должен вычислить сначала 'a' и только потом 'b'. Поэтому код вызывает неспецифицированное поведение.

Однако это не стоит путать с порядком вычисления самого выражения.

```
#include <iostream>
```

```
int f() {
    std::cout << 1;
    return 2;
}

int g() {
    std::cout << 2;
    return 3;
}

int main() {
    std::cout << f() + g() * g();
}
```

Результат выражения однозначно определяется и равен 11, однако результат выражения не связан с порядком вычисления f() и g(). f() и g() могли вычисляться в любом порядке, это не определено.

```
f(g(), h());
```

Выше также пример неспецифицированного поведения, g() и h() могут вычисляться в любом порядке.

Не все в стандарте специфицировано точно, потому что это не так важно, а некоторым компиляторам проще делать это по-своему.

Хочется добавить, что существуют некоторые полезные флаги компиляции.

Если посмотреть на рисунки 13 и 14, можно заметить, что clang++ отловил warning, а g++ нет. Если добавил флаг **-Wall**, то компилятор будет искать warning тщательнее. Флаг **-Wextra** это усилит.

```
iamnotexist@sasha:~/cpp/examples$ make l2-e13-wall-wextra
g++ -std=c++17 -Wall -Wextra l2/e13.cpp -o program
l2/e13.cpp: In function 'int main()':
l2/e13.cpp:14:9: warning: value computed is not used [-Wunused-value]
14 |     f() + g();
    |     ^~~~~~
```

Рис. 15: Компиляция g++ с -Wall и -Wextra.

По умолчанию warning не препятствуют компиляции, однако если написать флаг **-Werror**, каждый warning превратится в error, что не даст некорректной программе скомпилироваться.

```
g++ -std=c++17 -Wall -Wextra -Werror l2/e13.cpp -o program
l2/e13.cpp: In function 'int main()':
l2/e13.cpp:14:9: error: value computed is not used [-Werror=unused-value]
14 |     f() + g();
    |     ^~~~~~
ccplus: all warnings being treated as errors
make: *** [Makefile:59: l2-e13-wall-wextra-Error-1] Error 1
```

Рис. 16: Компиляция g++ с -Werror.

Благодаря этому мы сможем по максимуму избежать UB.

1.4. Declarations, definitions and scopes.

На самом деле наши предыдущие коды это последовательные объявления некоторых сущностей одной за другой, которые кончались main().

Формально сама программа это последовательность объявлений, с точки зрения компилятора.

Можно объявлять (вводить в рассмотрение) разные сущности. Например, переменные:

```
type id [= init];
```

Тип переменной, ее имя (идентификатор). Так же как необязательная часть возможна ее инициализация.

Также можно объявлять функции.

```
type fid(type1 id1, ..., typeN idN);
```

Тип функции, ее имя (идентификатор), типы и идентификаторы аргументов в круглых скобках через запятую.

Можно объявлять свои собственные типы.

```
struct S;  
class C;  
union U;
```

Можно объявлять пространства имен.

```
namespace N { }
```

Можно объявлять **alias** — другие названия чего-то объявленного (псевдонимы).

```
typedef shortname longTypeName;  
using shortname = longTypeName;
```

Компилятор будет транслировать shortname как longTypeName, что может сократить код.

Таким образом объявления **declarations** дают знать компилятору о каких-то сущностях. **Definitions** — определения сущностей.

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

В этом коде была объявлена переменная 'x'. Однако по стандарту она сразу же и определяется, в данном случае каким-то случайным значением, которое лежало на стеке.

```
int main() {  
    int x = 5;  
}
```

Здесь мы сами определяем переменную 'x' и даем ей значение 5.

С функциями все немного по-другому.

```
int g() {  
    f();  
}
```

```
int f() {  
    g();  
}
```

```
int main() {  
  
}
```

В этом коде хочется использовать функцию f() в определении функции g(), а функцию f() использовать в определении функции g(). Такой код не скомпилируется потому что на момент определения g() функция f() не объявлена.

```

int f();

int g() {
    f();
}

int f() {
    g();
}

int main() {

```

Здесь мы заранее объявили функцию `f()` поэтому код скомпилируется.

Таким образом для функции можно дать определение позже, но если мы хотим ее использовать, оно должно быть.

Существует некоторое правило: **One definition rule.**

Оно гласит в частности, что каждая функция и каждая переменная должна быть ровно один раз определена, если она не определена один раз, то ей нельзя пользоваться. Объявлять функцию можно сколько угодно раз.

```

int f();

int f();

int g() {
    f();
}

int f() {
    g();
}

int main() {

```

Поэтому такой код скомпилируется. Поскольку переменная одновременно определяется при создании такой трюк с ними уже не пройдет.

Лекция 3

Поговорим немного об областях видимости.

```

#include <iostream>

//глобальная область видимости

namespace N {
//область видимости namespace N
}

class C {
//область видимости класса C
};

int main() {
//область видимости функции
}

```

Область видимости (**scope**) — термин, который применяется к сущности и ограничивает ту часть программы, в которой эта сущность видна.

Если делать объявления в глобальной области видимости, то они будут видны отовсюду.

Началом scope будет являться открывающаяся фигурная скобка, концом — закрывающаяся.

Переменная, объявленная внутри scope функции называется **локальной**. А если переменная объявлена в глобальном scope то она **глобальная**.

```

#include <iostream>

```

```

int y;

int main() {
    int x;
}

```

В данном случае переменная 'y' глобальная, а 'x' — локальная.

Можно создавать **безымянные** scope.

```

#include <iostream>

```

```

int main() {
    int x;

    {
        int x;
    }
}

```

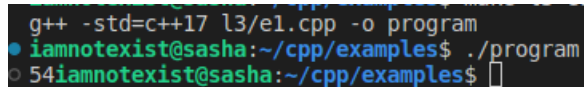
Обратим внимание на то, что можно объявлять переменные с одинаковым именем в разных scope.

```
#include <iostream>

int main() {
    int x = 4;

    {
        int x = 5;
        std::cout << x;
    }
    std::cout << x;
}
```

По умолчанию действия применяются к ближайшей переменной по вложенности к действию.



```
g++ -std=c++17 l3/e1.cpp -o program
iamnotexist@sasha:~/cpp/examples$ ./program
54iamnotexist@sasha:~/cpp/examples$
```

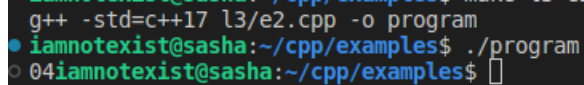
Рис. 17: Обращение к переменным в разных scope 1.

Существует способ обращения к переменным из глобального scope.

```
#include <iostream>

int x = 0;

int main() {
    int x = 4;
    std::cout << ::x;
    std::cout << x;
}
```



```
g++ -std=c++17 l3/e2.cpp -o program
iamnotexist@sasha:~/cpp/examples$ ./program
04iamnotexist@sasha:~/cpp/examples$
```

Рис. 18: Обращение к переменным в разных scope 2.

Глобальная переменная 'x' **затмевается** локальной переменной 'x' в scope функции main.

```
std::cout << ::x;
```

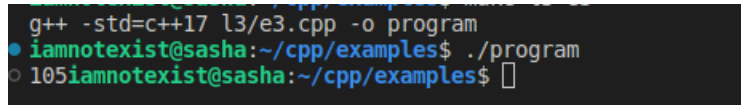
Написав '::' перед переменной мы говорим что хотим использовать 'x' из глобального scope.

```
#include <iostream>

namespace N {
    int x = 10;
}

int x = 5;

int main() {
    std::cout << N::x;
    std::cout << x;
}
```



```
g++ -std=c++17 l3/e3.cpp -o program
iamnotexist@sasha:~/cpp/examples$ ./program
105iamnotexist@sasha:~/cpp/examples$
```

Рис. 19: Обращение к переменным в разных scope 3.

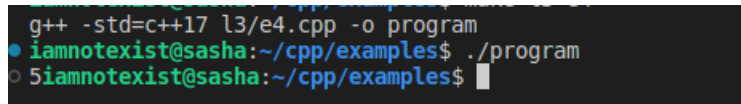
Чтобы обратиться к переменной из scope namespace N нужно писать 'N::'.

Таким образом **std** — это namespace, объявленный в стандартной библиотеке.

```
#include <iostream>

int main() {
    using namespace std;

    int x = 5;
    std::cout << x;
}
```



```
g++ -std=c++17 l3/e4.cpp -o program
iamnotexist@sasha:~/cpp/examples$ ./program
5iamnotexist@sasha:~/cpp/examples$
```

Рис. 20: Вливание std.

Написав `using namespace std` мы сливаем namespace std со скоупом функции main и можем обращаться к cout без префикса std::.

Делать подобное не рекомендуется поскольку std довольно большой и велик риск коллизии объявлений.

Влить в scope функции можно и что-то конкретное.

```
using std::cout;
```

В C++ есть некоторые понятия. Если переменная используется без префикса scope, то это **unqualified-id**, а если с префиксом то это **qualified-id**.

1.5. Basic types and supported operation.

C++ — статически типизированный язык. Это значит, что у каждой переменной должен быть тип и он должен быть известен на этапе компиляции.

У переменной нельзя менять ее тип.

Приведем примеры базовых целочисленных типов:

- short - обычно 2 байта, $[-2^{15}; 2^{15})$
- int - обычно 4 байта, $[-2^{31}; 2^{31})$
- long - обычно 4 байта, $[-2^{31}; 2^{31})$
- long long - обычно 8 байт, $[-2^{63}; 2^{63})$

Помимо этого у них есть беззнаковые версии:

- unsigned short - обычно 2 байта, $[0; 2^{16})$
- unsigned int - обычно 4 байта, $[0; 2^{32})$
- unsigned long - обычно 4 байта, $[0; 2^{32})$
- unsigned long long - обычно 8 байт, $[0; 2^{64})$

Также в программе можно написать просто unsigned. По умолчанию будет считаться, что это unsigned int. Размер этих типов не определен стандартом. Однако иногда требуется точность, поэтому существуют следующие типы.

- int16_t - 2 байта, $[-2^{15}; 2^{15})$
- int32_t - 4 байта, $[-2^{31}; 2^{31})$
- int64_t - 8 байт, $[-2^{63}; 2^{63})$
- uint16_t - 2 байта, $[0; 2^{16})$
- uint32_t - 4 байта, $[0; 2^{32})$
- uint64_t - 8 байт, $[0; 2^{64})$

Существует также тип **size_t**, который является alias для некоторого беззнакового типа. То какого типа это alias зависит от размера адресного пространства.

Перейдем к вещественным типам.

- float - обычно 4 байта
- double - обычно 8 байт
- long double - обычно 16 байт

Рассмотрим по подробнее double.

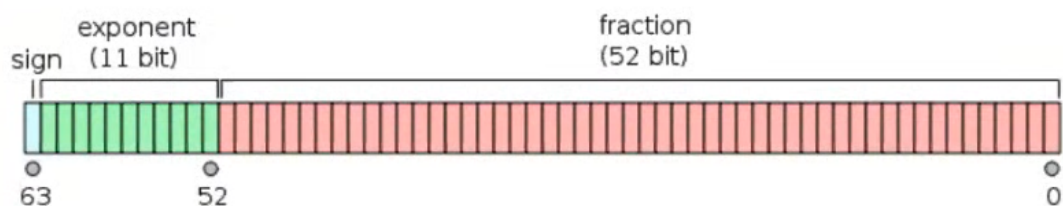


Рис. 21: Представление double.

1 старший бит отводится под знак. 11 бит под экспоненту. 52 бита под мантиссу.

В экспоненте записано число — степень, в которую нужно возвести двойку. В мантиссе информация о втором множителе.