3.1-3.2 Понятие компилятора. Примеры компиляторов C++. Понятие ошибок компиляции (CE) с примерами. Понятия ошибок времени выполнения (RE) и неопределенного поведения (UB) с примерами. Почему не любую RE компилятор может предвидеть? Почему не любое UB приводит к RE?

Понятие компилятора

Компилятор – это программа, которая переводит текст, написанный на языке программирования, в набор машинных кодов.

- Компилируемые языки:
  * Программа на компилируемом языке при помощи специальной программы компилятора преобразуется (компилируется) в набор инструкций для данного типа процессора (машинный код) и далее записывается в *исполняемый файл*, который может быть запущен на выполнение как отдельная программа. Другими словами, компилятор переводит программу с языка высокого уровня на низкоуровневый язык, понятный процессору сразу и целиком, создавая при этом отдельную программу

  Примерами компилируемых языков являются Pascal, C, C++, Rust, Go.

- Интерпретируемые языки
  * Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) ее текст без предварительного перевода. При этом программа остается на исходном языке и не может быть запущена без интерпретатора. Можно сказать, что процессор компьютера — это интерпретатор машинного кода. Кратко говоря, интерпретатор переводит на машинный язык прямо во время исполнения программы.

  Примерами интерпретируемых языков являются PHP, Perl, Ruby, Python, JavaScript. К интерпретируемым языкам также можно отнести все скриптовые языки.

Note: Java и C, находятся между компилируемыми и интерпретируемыми. А именно, программа компилируется не в машинный язык, а в машинно-независимый код низкого уровня, байт-код.

Примеры компиляторов C++:

1. GNU Compiler Collection aka GCC
2. Clang
3. C++ Builder
4. Microsoft Visual C++

Запуск компилятора из командной строки: **g++ main.cpp** или **clang++ main.cpp**
Запуск исполняемого файла из командной строки: **./a.out**

Compile time error aka CE

Ошибка времени компиляции возникает, когда код написан некорректно с точки зрения языка. Из такого кода не получается создать исполняемый файл. Примеры:

1. **Лексические:** ошибка в процессе разбиения на токены, т.е. компилятор увидел последовательность символов, которую не смог расшифровать.

   ✓ `(std) (::) (cout) (<<) (x) (;)` – пример корректного разбития на токены

   × `24abracadabra;`

2. **Синтаксическая:** возникает, когда вы пишете инструкцию, недопустимую в соответствии с грамматикой языка (например служит речь Мастера Йоды)

   × `int const x = 5;`

   × `x + 5 +;`

   × нет точки с запятой (;)

   × несоответствие круглых или фигурных скобок

3. **Семантическая:** возникате, когда инструкция написана корректно, но компилятор ее выполнить не может (например: съешьте себя этим столом)

   × использование необъявленных переменных

   × вызов метода `size()` от переменной типа int

   × `x++ = a + b;`

   × вызов `foo(3);` хотя сигнатура такая: `void foo(int a, int b);`

Runtime error aka RE

Программа компилируется корректно, но в ходе выполнения она делает что-то непотребное. RE невозможно отследить на этапе компиляции (компилятор может разве что кинуть предупреждение в месте потенциальной ошибки). Примеры:

× Слишком большая глубина рекурсии – **stack overflow ⇒ segmentation fault**

× Слишком далекий выход за границу массива – **segmentation fault**

× Целочисленное деление на ноль (не всегда компилятор такое может предвидеть)

× Исключение, которое никто не поймал – RE

Замечание: не всякое исключение есть RE, и не каждое RE есть исключение.

Undefined behaviour aka UB

UB возникает при выполнении кода, результат исполнения которого не описан в стандарте. В случае UB компилятор волен сделать, всё что угодно, поэтому результат зависит от того, чем и с какими настройками код был скомпилирован (в теории компилятор может взорвать компьютер). UB может переродиться в CE, RE, или пройти незамеченным и нормально отработать. Примеры:

× Для **static_cast** преобразование указателя на родительский класс к указателю на дочерний класс. Объект по указателю обязан быть правильного дочернего класса, иначе это undefined behaviour.

× Битые ссылки.

× `++x = x++;` или `f(x=y, x=3);` — порядок вычисления аргументов оператора и функций не определён. (until C++17)

× `int x = 2 << 40;` – не определено, что будет происходить при переполнении знакового типа.

× Чтение выделенной, но неинициализированной памяти. В теории, считается какой-то мусор, но технически, так как это UB, компилятор в праве поджечь ваш монитор.

× Отсутствие **return** в конце функции, которая что-то возвращает. Шок, да? Это UB!

× Недалекий выход за границы **C-style массива**.

Замечание: К сожалению, математически нельзя сделать все ошибки CE. За счёт UB в C++ мы выигрываем в эффективности.

3.3-3.4 Основные типы. Приоритет операций.

C++ является статически-типизированным языком, то есть на момент компиляции все типы должны быть известны. Основные типы, с которыми мы сталкивались:

Тип	*байт*	*Диапазон значений*
логический тип данных		
bool	1	$[0; 2^8)$
целочисленные типы данных		
short	2	$[-2^{15}; 2^{15})$
unsigned short	2	$[0, 2^{16})$
int	4	$[-2^{31}; 2^{31})$
unsigned int	4	$[0; 2^{32})$
long long	8	$[-2^{63}; 2^{63})$
unsigned long long	8	$[0; 2^{64})$
вещественные типы данных		
float	4	$[-2^{31}; 2^{31})$
double	8	$[-2^{63}; 2^{63})$
long double	10	.....
символьные типы данных		
char	1	$[-2^7; 2^7)$
unsighned char	1	$[0; 2^8)$

Так же используются **литеральные суффиксы:**

- **.u** для unsigned int
- **.ll** и **.ull** для long long и unsigned long long
- **.f** для float
- и прочее

Integer promotion: ”меньший” тип приводится к ”большему”

Замечание: Неявное преобразование к unsign: может вызывать проблемы, например:

```
int x = -1;
unsigned y = 0;
std::cout << x + y;  // very big number
```

Note: size_t беззнаковый целый тип данных, возвращаемый оператором sizeof, определен в заголовочном файле `<cstring>`

Note: Сложение char - это UB

Выражения и операторы

Идентификаторы — любая последовательность латинских букв, цифр и знака “_”, не начинающаяся с цифры. Они не могут совпадать с ключевыми словами (new, delete, class, int, if, true, etc)

Литералы — последовательность символов, интерпретируемая как константное значение какого-то типа (1, ‘a’, “abc”, 0.5, true, nullptr, etc)

Операторы — это, можно сказать, функции со специальными именами (`=, +, <, [], ()`)

Выражение — некоторая синтаксически верная комбинация литералов и идентификаторов, соединенных операторами

В следующей таблице перечислены приоритет и ассоциативность операций C++. Операции перечислены сверху вниз в порядке убывания приоритета.

Приоритет	Оператор	Описание	Ассоциативность
1	`::`	Разрешение области видимости	слева направо
2	`a++ a--`	Суффиксный/постфиксный инкремент и декремент (возвращает копию старого объекта)	
	тип`()` *тип*`{}`	Функциональный оператор приведения типов	
	`a()`	Вызов функции	
	`a[]`	Индексация	
	`. ->`	Доступ к полю или методу класса (через объект класса / указатель на объект)	
3	`++a --a`	Префиксный инкремент и декремент (возвращает ссылку на уже измененный объект)	Справа налево
	`+a -a`	Унарные плюс и минус	
	`! ~`	Логическое НЕ и побитовое НЕ	
	(*тип*)	Приведение типов в стиле C	
	`*a`	Косвенное обращение (разыменование)	
	`&a`	Взятие адреса	
	`sizeof`	Размер в байтах[примечание 1]	
	`co_await`	Выражение await (C++20)	
	`new new[]`	Динамическое распределение памяти	
	`delete` `delete[]`	Динамическое освобождение памяти	
4	`.* ->*`	Указатель на элемент	Слева направо
5	`a*b a/b` `a%b`	Умножение, деление и остаток от деления	
6	`a+b a-b`	Сложение и вычитание	
7	`<< >>`	Побитовый сдвиг влево и сдвиг вправо	
8	`<=>`	Оператор трёхстороннего сравнения (начиная с C++20)	
9	`< <=`	Операторы сравнения < и ≤ соответственно	
	`> >=`	Операторы сравнения > и ≥ соответственно	
10	`== !=`	Операторы равенства = и ≠ соответственно	
11	`&`	Побитовое И	
12	`^`	Побитовый XOR (исключающее или)	
13	`|`	Побитовое ИЛИ (включающее или)	
14	`&&`	Логическое И	
15	`||`	Логическое ИЛИ	
16	`a?b:c`	Тернарный условный оператор[примечание 2]	Справа налево
	`throw`	Оператор throw	
	`co_yield`	Выражение yield (C++20)	
	`=`	Прямое присваивание (предоставляется по умолчанию для классов C++)	
	`+= -=`	Составное присваивание с сложением и вычитанием	
	`*= /= %=`	Составное присваивание с умножением, делением и остатком от деления	
	`<<= >>=`	Составное присваивание с побитовым сдвигом влево и сдвигом вправо	
	`&= ^= |=`	Составное присваивание с побитовым И, XOR и ИЛИ	
17	`,`	Запятая (возвращает то, что справа)	Слева направо

При синтаксическом анализе выражения оператор, указанный в некоторой строке приведённой выше таблицы, будет более тесно связан со своими аргументами (как в случае применения скобок), чем любой оператор из строк, расположенных ниже, с более низким приоритетом. Например, выражения `std::cout<<a&b` и `*p++` будут разобраны как `(std::cout<<a)&b` и `*(p++)`, а не как `std::cout<<(a&b)` и `(*p)++`.

Операторы с одинаковым приоритетом связываются со своими аргументами в направлении их ассоциативности. Например, выражение `a = b = c` будет разобрано как `a = (b = c)`, а не `(a = b) = c`, так как ассоциативность присваивания справа налево, но `a + b - c` будет разобрано как `(a + b) - c`, а не `a + (b - c)`, так как ассоциативность сложения и вычитания слева направо.

Использования оператора запятая

```
    int x = 0;
    int y = 2;
    int z = (++x, ++y);
    std::cout << z;  // it will be 3

    // BUT!
    int sum = add(x, y); // this is not a comma operator
    int x=3, y=5; // this one also
```

Использования тернарного оператора

Тернарный оператор выделяется из ряда других операторов в C++. Его называют *"conditional expression"*. Ну а так как это *expression*, выражение, то как у каждого выражения, у него должен быть **тип** и **value category**.

Типом тернарного оператора будет **наиболее общий тип** его двух последних операндов. (Например, у int и short общим типом будет int.) Т.е. наиболее общий тип это такой тип, к которому могу быть приведены оба операнда. Вполне могут быть ситуации, когда общего типа нет.

```
    struct C{};
    struct D{};
    (true ? C() : D());  // this will cause CE
```

Так. С типом тернарного оператора мы немного разобрались. Осталось решить вопрос с value category. Тут действует следующее **правило:** если в тернарном операторе происходит преобразование типов к наиболее общему, то тернарный оператор — *rvalue*, иначе *lvalue*.

```
    int i;
    int j;
    (true ? i: j) = 45;  // OK

    short i;
    int j;
    (true ? i: j) = 45; // CE
    // Здесь происходит преобразование типов.
    // Значит value category у выражения слева от знака -"- rvalue.
    // A rvalue, как известно, нельзя присваивать.


    int a = 1;
    int b = 1;
    int c = 1;
    a = true ? ++b : ++c;
    // a = 2, b = 2, c = 1
```

Где нельзя использовать if{...} else{...}, но можно тернарный оператор?

Например, в списке инициализации конструктора и при инициализации ссылки в зависимости от условия. (Как известно, нельзя объявлять не инициализированную ссылку)

```
    // 1-ый пример
    struct S {
        int i_;

        // вот так НЕЛЬЗЯ, это CE
        S() : if(some_condition) i_(1) else i_(0) {}

        // а вот так можно :)
        S() : i_(some_condition ? 1 : 0) {}
    };



    // 2-ой пример
    int a = 3, b = 4;
    int& i;

    // вот так НЕЛЬЗЯ, это CE
    if(some_condition) i = a;
    else i = b;

    // а вот так можно :)
    int& i = (some_condition ? a : b);
```