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!

× 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 Основные типы и операции над ними

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 / 255
целочисленные типы данных		
short	2	-32 768 / 32 767
unsigned short	2	0 / 65 535
int	4	-2 147 483 648 / 2 147 483 647
unsigned int	4	0 / 4 294 967 295
long long	8	-2^{63} / $2^{63}-1$
unsigned long long	8	0 / $2^{64}-1$
вещественные типы данных		
float	4	-2 147 483 648.0 / 2 147 483 647.0
double	8	-2^{63} / $2^{63}-1$
long double	10	.....
символьные типы данных		
char	1	-128 / 127
unsighned char	1	0 / 255

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 в сумме очень большое число.

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

Выражения и операторы

1. **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)

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

3. **Операторы** — это, можно сказать, функции со специальными именами (=, +, <, [], (), etc)

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

5. **Тернарный оператор** (?:). “Условие” ? “выражение, если true” : “выражение, если false”

6. **Оператор “запятая”** (,). Вычисляет то, что слева, затем вычисляет то, что справа и возвращает то, что справа. (Имеет самый низкий приоритет)

7. **Унарная “звёздочка”** (*). Разыменование

8. **Унарный “амперсанд”** (&). Взятие адреса

9. **Оператор “точка”/”стрелочка”**. Доступ к полю/методу класса (соответственно через объект класса/указатель на объект)

10. **Двойное двоеточие**. Переход в другую область видимости (std::cout, ::operator new)

11. **Префиксный/постфиксный инкремент**. Префиксный увеличивает на единицу и возвращает ссылку на уже измененный объект. Постфиксный увеличивает на единицу и возвращает копию старого объекта.

12. **Бинарный амперсанд**. Побитовое И

13. **Бинарный двойной амперсанд**. Логическое И

14. **Оператор присваивания**. Присваивает значение (копированием или перемещением)

15. **Оператор составного присваивания**. Легче пример: $a+ = 5 \Leftrightarrow a = a + 5$. Только во втором случае создается лишняя копия

16. **Оператор** $<<$ **(**$>>$**)**. В заивисмости от контекста это либо побитовое смещение влево (вправо), либо это оператор ввода(вывода) в(из) поток(а).

3.4 В каком порядке вычисляются операторы над основными типами.

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right

2	`a++` `a--`
	`type``()` *`type`*`{}`
	`a()`
	`a[]`
	`.` `->`
		Suffix/postfix increment and decrement
		Functional cast
		Function call
		Subscript
		Member access	

3	`++a` `--a`
	`+a` `-a`
	`!` `~`
	`(`*`type`*`)`
	`*a`
	`&a`
	`sizeof`
	`co_await`
	`new` `new[]`
	`delete` `delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		C-style cast
		Indirection (dereference)
		Address-of
		Size-of[note 1]
		await-expression (C++20)
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left

4	`.*` `->*`	Pointer-to-member	Left-to-right
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`<` `<=` `>` `>=`	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
10	`==` `!=`	For equality operators = and ≠ respectively	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	

16	`a?b:c`
	`throw`
	`co_yield`
	`=`
	`+=` `-=`
	`*=` `/=` `%=`
	`<<=` `>>=`
	`&=` `^=` `|=`
		Ternary conditional[note 2]
		throw operator
		yield-expression (C++20)
		Direct assignment (provided by default for C++ classes)
		Compound assignment by sum and difference
		Compound assignment by product, quotient, and remainder
		Compound assignment by bitwise left shift and right shift
		Compound assignment by bitwise AND, XOR, and OR
			Right-to-left

17	`,`	Comma	Left-to-right