Содержание

1 Введение в язык C++

1.1 История Языка C++

- C (1969)

  Прародитель языка C++, был создан для упрощения реализации низкоуровневых программ

- C with classes (1979)

  Бьерн Страуструп (Bjarne Stroustrup) решил создать свой языкпрограммирования, который бы сочетал в себе скорость низкоуровневыхязыков и удобство разработки на высокоуровневых.

- C++ (1982)

- C++98 (1998)

  Первый официальный стандарт языка C++

- C++03, C++11, C++14, C++17, C++20, ...

1.2 Современный язык C++

- Неизменно в топе самых популярных и востребованных языков
- C++ используется в ситуациях, когда важна скорость. В частности, приразработке операционных систем, баз данных, серверных и облачныхприложений, утилитах для работы с графикой.
- Обширный набор низкоуровневых инструментов позволяет эффективноиспользовать C++ во встраиваемых системах.
- Что написано на C++? — *Windows, OS X, Google Chrome, Mozila Firefox, YouTube, Photoshop, Telegram*, другие языки программирования ...

1.3 Состав языка C++

1.3.1 Алфавит

Любая программа на языке C++ состоит из следующих символов:

- Цифры (`0` , `1` , ... , `9`)
- Буквы латинского алфавита (`a` , `b` , ... , `z` , `A` , ... , `Z`)
- Нижнее подчеркивание `_`
- Пробельные символы и перенос строки
- Скобки (`{}` , `()` , `[]` , `<>`)
- Кавычки и апострофы (`"` , `'`)

1.3.2 Ключевые слова

Символы могут составлять слова, часть из которых имеют особое значение.

Слова с особым значением названются *ключевыми словами*.

`alignas` , `alignof` , `and` , `and_eq` , `asm` , `auto` , `bitand` , `bitor` , `bool` , `break` , `case` , `catch` , `char` , `char8_t` , `char16_t` , `char32_t` , `class` , `compl` , `concept` , `const` , `consteval` , `constexpr` , `constinit` , `const_cast` , `continue` , `co_await` , `co_return` , `co_yield` , `decltype` , `default` , `delete` , `do` , ...

Полный список ключвых слов можно найти по ссылке

1.3.3 Идентификаторы

- *Идентификатор* — имя программного объекта.
- Идентификатор представляет собой последовательность букв, цифр, нижнихподчеркиваний, составленную по следующим правилам:

1. Не начинается с цифры
2. Не содержит пробелов
3. Не является ключевым словом
4. Чувствительно к регистру (`Alice` и `alice` — разные идентификаторы)

Примеры:

```
alla                //OK
allo4ka             //OK
allochka-ivanova    //CE (character '-' cannot be in identifier)
allo4ka_ivanova     //OK
allochka ivanova    //CE (must not contain whitespace characters)
2dima               //CE (must not start with a number)
_                   //OK
_2dima              //OK
asm                 //CE (key word)
AsM                 //OK
```

1.3.4 Константы/литералы

Константы/литералы — значения встроенные в текст программы.

- Целочисленные константы:
  - Десятичные `123`
  - Восьмеричные `045`
  - Шестнадцатеричные `0x1A` , `0XBD`
  - Двоичные (C++14) `0b101` , `0B1001`
- Символьные константы — `'a'` , `'n'`

- Вещественные константы — `3.14` , `.24e10`
- Строковые литералы: `"a"` , `"c"` , `"n"` , `"This␣is␣string"`
- Логические константы — `true` , `false`
- Нулевой указатель — `nullptr`

1.3.5 Знаки операций

Каждая операция(operator) характеризуется *арностью, приоритетом и ассоциативностью.*

- Арность — количество аргументов, которое может принимать операций.
- Приоритет — свойство, которое определяет порядок вычисления операций.
- Ассоциативность — свойство, которое определяет порядок выполнения операций с одинаковым приоритетом

Важно: Один и тот же знак операции может иметь разный смысл, арность и приоритет в зависимости от количества операндов, типа данных.

Например:

```
10 / 3;    //Integer division - 3
-10 / 3.; //Real division - 3.3333
-5;        //Unary operation
5 - 3;     //Binary operation
```

Более подробно про операции можно прочитать по ссылке

1.4 Запуск программ на C++

- Язык C++ — *компилируемый* язык. Это означает, что код (текст программы) сначала обрабатывается специальной программой (компилятор), которая переводит его в машинные инструкции и создает исполняемый файл. Далее этот файл может быть запущен и исполнен как любая другая программа.

  Примерами таких языков являются — *C, C++, Basic, Go, Pascal, Rust, ...*

- Также существуют *интерпретируемые* языки программирования. В них текст программы подается на вход другой программе (интерпретатор), который сразу начинает исполнение написанного кода без этапа сборки исполняемого файла.

  Примерами таких языков являются — *Python, PHP, Perl, Ruby*

- В некоторых языках (например, *Java*) используется гибридный подход.

Кодом на C++ является любой текстовый файл, написанный по правилам языка C++.

Чтобы сделать из него программу, необходимо его *скомпилировать*, то есть передать код программе-компилятору.

После этого появится программа (файл), которую можно запустить с помощью командной строки:

```
g++ <files name> -o <programms name>
```

1.5 Первая программа

Минимальная программа, которая ничего не делает (почти), на C++ выглядит так:

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

Данная программа состоит из ключевого слова (`int`), идентификатора (`main`), скобок (`()` , `{}`)

Пояснение: здесь происходит определение функции с именем `main` , которая ничего не принимает(`()`), возвращает целое число(`int`) и не делает ничего (`{}`). Функция с таким именем обязана присутствовать в любой программе, так как именно с нее начинается исполнение программы.

Функция `main` по умолчанию возвращает `0` . Но это значение можно указать явно:

```
int main() {
  return 0; // Key word return + constant (literal) 0
}
```

Кроме того, мы можем выполнять целочисленную арифметику в C++ и возвращать ее результат.

```
int main() {
  return 2 + 2 ; 2;
}
```

Примечание: сначала вычисляется результат выражения справа от `return` , а затем это значение возвращается из функции `main` .

1.6 Структура программы

1.6.1 Объявления

Любая программа на C++ — это последовательность *объявлений.*

Объявление (declaration) — введение (создание) некоторой именованной сущности.

Пример:

```
int main() { // this is a function declaration "main"
  return 0;
}
```

Объявление функции `main` можно сократить:

```
int main();
```

Такое объявление является объявлением без определения.

1.6.2 Определение

В общем случае объявления лишь вводят новые имена, но не дают им конкретного смысла.

Определение (definition) — объявление с полной информацией о создаваемой сущности.

Пример:

```
int main(); // This is the function declaration
int main() { // this is the function definition
  return 0;
}
```

Важно: в программе должно быть ровно одно определение функции `main` . Нельзя не определять `main` .

1.6.3 Директива

Помимо объявлений в программе могут встречаться директивы препроцессора.

Директивы препроцессора — вспомогательные инструкции, которые начинаются с `#` и осуществляют манипуляции над текстом программы и опциями сборки

Примеры:

```
/* Paste the text file "file" from the current folder to the
given location in the code */
#include "file"
/* Paste the text file "file" from a specific folder at the
given location in the code*/
#include <file>
// Replace all occurrences of text "X" with text "Y"
#define X Y
```

1.6.4 Операторы (инструкции)

Функции (в частности, функция `main`) описывают последовательность действий, которые выполняются по порядку, сверху-вниз.

Оператор (инструкция, statement) — фрагмент кода, описывающий некоторое действие.

На данный момент мы знакомы с одним оператором — оператором возврата из функции `return` .

1.6.5 Выражения

Некоторые действия (операторы) требуют проведения вычислений. За вычисления в языке C++ отвечают *выражения.*

Выражение (expression) — последовательность операций и их операндов, задающая вычисление.

У любого выражения есть результат (может быть пустой), который "подставляется"вместо него.

```
int main() {
  return 2 + 2 * 2; // 2 + 2 * 2 - expression with result 6.
}
```

1.6.6 Комментарии

Иногда в коде хочется сделать некоторое примечание или напоминание о том, что означает тот или иной блок кода. Но мы знаем, что просто так написать текст в коде нельзя, так как,

скорее всего, он не будет удовлетворять правилам грамматики языка C++. Для этих целей в языке есть *комментарии*.

Комментарий — текст кода, который игнорируется при сборке программы.

В C++ есть два типа комментариев:

```
// single line comment
/* multiline
comment */
```

1.7 Вывод результата в консоль

Все программы, которые были написаны нами ранее, возвращали результат операционной системе в виде "кода ошибки":

```
int main() {
  return 2 + 2 * 2;
}
```

Это совсем неудобно, так как

1. Результатом работы программы не всегда является целое число
2. Коды ошибок могут принимать лишь значения от 0 до 255
3. Результат приходится получать отдельно с помощью специальной переменной окружения — `echo $?`

Для вывода результата на экран в C++ обычно используется *потоковый вывод*:

```
#include <iostream> // 1
int main() {
  std::cout << 2 + 2 * 2; // 4
  return 0; // 0 - no errors
}
```

- строка 1:

  Знаем, что `#include` — директива, означающая "вставку"другого файла в текст нашей программы. В файле `iostream` (Input-Output STREAM) сожержится все необходимое для работы с выводом.

- строка 4: В этой строке записано **выражение** вывода результата `2 + 2 * 2` в поток `std::cout` (Console OUTput).

В поток вывода можно записывать и другую информацию, например, строки. Хорошим тоном является вывод переноса строки в конце результата (для более красивого сообщения на экране). За это отвечает специальный символ `'n'` .

Пример:

```cpp
#include <iostream>
int main() {
  std::cout << "2 + 2 * 2 = ";
  std::cout << 2 + 2 * 2;
  std::cout << '\n';
  return 0;
}
```

Весь вывод можно объединить в одно выражение:

```cpp
#include <iostream>
int main() {
  std::cout << "2 + 2 * 2 = " << 2 + 2 * 2 << '\n';
  return 0;
}
```

1.8 Самая важная программа

Используя накопленные знания, напишем программу, приветствующую мир:

```cpp
#include <iostream>
int main() {
  std::cout << "Hello, world!\n";
  return 0;
}
```

Примечание: обратите внимание, что символ переноса строки может стоять внутри строки, не обязательно выводить его отдельно

2 Фундаментальные типы данных

2.1 Типы данных

Тип данных — свойство программных сущностей, которое определяет:

1. Множество допустимых значений
2. Допустимые операции
3. Низкоуровневое представление (интерпретация битов)

В C++ существует большое количество типов данных. Более того, можно (и мы будем) создавать и свои типы. Однако каждый тип данных состоит из, или является производным от одного из фундаментальных типов:

- "Пустой"тип — `void`
- Целые тип — `int` , `short` , `long` , `long long`
- Числа с плавающей точкой — `float` , `double` , `long double`

- Символьный тип — `char` , `char8_t` , `char16_t` , `char32_t`
- Логический тип — `bool`
- Нулевой указатель — `std::nullptr_t`

2.2 Целый тип данных

Самым используемым типом данных является целочисленный тип. Поэтому ему мы уделим больше всего внимания.

2.2.1 int

Основным целочисленным типом является `int` .

Стандарт C++ гарантирует, что размер `int` **как минимум** 16 бит (2 байта).

Это значит, что значения типа `int` могут лежать в диапазоне $[-32'768; 32'767]$.

На практике, как правило, `int` занимает 32 бита (4 байта), то есть принимает значения в диапазоне $2'147'483'648; 2'147'483'647$

.

Литералами типа `int` являются — 0, 1, -4, 100, $576'843$

2.2.2 long

Стандарт C++ гарантирует, что размер `long` **как минимум** 32 бита (4 байта) и **не меньше** размера `int` .

Это значит, что значения типа `long` могут лежать в диапазоне $[2'147'483'648; 2'147'483'647]$.

Литералами типа `long` являются — $0l$, $1L$, $-4l$, $100L$, $1'123'576'843l$

2.2.3 long long

Стандарт C++ гарантирует, что размер `long long` **как минимум** 64 бита (8 байт) и **не меньше** размера `long` .

Это значит, что значения типа `long long` могут лежать в диапазоне $[9'223'372'036'854'775'808; 9'223'372'036'854'775'807]$.

Литералами типа `long long` являются — $0ll$, $1LL$, $-4ll$, $100LL$, $1'123'576'843ll$

2.2.4 short

Стандарт C++ гарантирует, что размер `short` **как минимум** 16 бит (2 байта) и **не больше** размера `int` .

Это значит, что значения типа `short` могут лежать в диапазоне $[-32'768; 32'767]$.

Собственных литералов тип `short` не имеет.

2.2.5 Модификаторы signed/unsigned

По умолчанию все целые типы *знаковые*, то есть могут хранить как положительные, так и отрицательные числа. Чтобы это подчеркнуть, к названиям типов можно приписывать модификатор `signed` :

```
int == signed int == signed
long == signed long == long int == signed long int
long long == signed long long == long long int == signed long long int
short == signed short == short int == signed short int
```

Если предполагается, что значение должно хранить только неотрицательные числа, то можно к имени типа добавить `unsigned` .

В этом случае допустимые значения меняются следующим образом:

- 2 байта — $[0; 65'535]$
- 4 байта — $[0; 4'294'967'295]$
- 8 байт — $[0; 18'446'744'073'709'551'615]$

```
unsigned int == unsigned
unsigned long == unsigned long int
unsigned long long == unsigned long long int
unsigned short == unsigned short int
```

Беззнаковые литералы — $1u$, $4ul$, $6ull$

2.2.6 Операции

Над целыми числами (кроме short !) можно выполнять все арифметические операции. Правила выполнения операций звучат следующим образом:

- Результатом операции является значение того же типа, что и у операндов
- При переполнении беззнаковых чисел выполняется арифметика по модулю размера этого типа
- В остальных ситуациях переполнение — это *Undefined Behavour*(неопределенное поведение)
- Деление на 0 — *Undefined Behavour*
- Битовые сдвиги отрицательных чисел тоже могут приводить к UB

Если же операнды имеют разные типы (или тип `short`), то:

- (`signed` / `unsigned`) `short` приводится к `int`
- Менее широкий тип приводится к более широкому
- Знаковый тип приводится к беззнаковому

Примеры:

```
0 + 0l; // long
0ll + 0; // long long
0l + 0ll; // long long
0u + 0; // unsigned int
0 + 0ul; // unsigned long
0ul + 0ll; // unsigned or signed long long
```

2.2.7 Целые типы фиксированной ширины

На разных системах целые типы могут иметь разные размеры, что осложняет жизнь.

Для решения этой проблемы в C++11 появились типы *фиксированной ширины*:

```
int8_t, int16_t, int32_t, int64_t
uint8_t, uint16_t, uint32_t, uint64_t
```

Они имеют **в точности** тот размер, который указан в названии. Чтобы использовать их, необходимо подключить заголовочный файл `<cstdint>`

2.3 Символьные типы

Символьные типы данных используются со строками. Они будут рассмотрены далее в курсе, но пока необходимо с ними познакомиться.

2.3.1 char

Тип `char` используется для хранения символов.

Символ представляется некоторым 1 байтовым целым числом согласно ASCII таблице

Символьные литералы — `'a'`, `'1'`, `'n'`, `'@'`, ...

Так как `char` — целое число, оно может быть как знаковым, так и беззнаковым (какой из типов используется по умолчанию - зависит от реализации).

2.3.2 char_t, char16_t, char32_t

Широкие символьные типы используются для хранения символов из кодировок $UTF-16$ и $UTF-32$. Более подробно по ссылке.

2.3.3 Арифметика

Так как символы представляются целыми числами, тип char можно считать 8-битным целым числом и использовать все арифметические операции.

`'E' + ('a' - 'A') == 'e'`

Напоминание: при выполнении арифметики числа, у которых тип имеет ранг меньший int , приводятся к int .

Примеры:

```
'a' + 0; // int
'a' + 0l; // long
'a' + 0ll; // long long
'a' + 'a'; // int
```

2.4 Логический тип

2.4.1 bool

Объекты логического типа могут принимать всего 2 значения: `true` / `false` .

Имеет размер в 1 байт.

Обычно используется для хранения результата сравнения:

```
5 > 6;  // false
5 < 6;  // true
5 >= 6; // false
5 <= 6; // true
5 == 6; // false
5 != 6; // true
```

Объекты типа `bool` могут выступать операндами логических операций:

```
5 > 6 && 5 < 6; // false (logical "and")
5 > 6 || 5 < 6; // true (logical "or")
!(5 > 6); // true (logical "not")
```

Также в C++ возможно использование специальных слов `and` , `or` , `not` . Но это является плохим стилем, так как эти ключевые слова пришли с языка C, где первоначально не было специальных символов. Использовать их не рекомендуется, а на нашем курсе — запрещено.

`&&` и `||` — особенные операции:

1. Гарантируется, что выражение слева будет вычислено до выражения справа.
2. Если слева значение `true` , то правая часть `||` вычисляться не будет.
3. Если слева значение `false` , то правая часть `&&` вычисляться не будет.

```
5 < 6 || ... ; // whatever is on the right is not evaluated (at all)
5 > 6 && ...; // whatever is on the right is not evaluated (never)
```

2.4.2 Арифметика

Логический тип в C++ является разновидностью целового типа и может быть использован в арифметических выражениях. При этом `true == 1` , а `false == 0` :

```
true + 5; // int: 6
10l * false; // long: 0
```

Верно и обратное: при подстановке в логическую операцию ненулевое значение интерпретируется как `true`, нулевое — `false`:

```
5 && 1; // true
0 || 0; // false
!-1;    // false
```

2.5 Числа с плавающей точкой

Числа с плавающей точкой используются для хранения рациональных чисел.

`float` — числа с одинарной точностью (4 байта, примерно $[\pm 10^{-38}; \pm 10^{38}]$)

`double` — числа с двойной точностью (8 байт, примерно $[\pm 10^{-308}; \pm 10^{308}]$)

`long double` — числа с расширенной точностью (12 или 16 байт, примерно $[\pm 10^{-4932}; \pm 10^{4932}]$)

Вещественные литералы:

```
0., 1.5, 3.14159; // double
0f, 1.5F, 3.14159f; // float
0l, 1.5L, 3.14159l; // long double
123.456e10; // 123.456 * 10^10
123.456e-10; // 123.456 * 10^(-10)
```

2.5.1 Особенности

- Можно применять те же арифметические операции, что и к целым числам (кроме битовых операций и взятия остатка)

  `0.1 + 0.2 * 5.67 / 0.9; //real division`

- Следует помнить, что дробные числа имеют ограниченную точность при вычислениях:
  `0.1 + 0.2 != 0.3;`

- Числа с плавающей точкой всегда знаковые.

- Имеются специальные значения: `+inf`, `-inf`, `nan`:

  ```
  1. / 0;  //inf
  -1. / 0; //-inf
  0. / 0;  //nan
  ```

- Если типы чисел с плавающей точкой не совпадают, то менее широкий аргумент приводится к более широкому:

  ```
  5.0 + 1.5f; // double
  5.0 + 1.5l; // long double
  ```

- При выполнении арифметической операции над целым и дробным числом целое число приводится к типу дробного:

  `1 + .0f; //float`

2.6 Пустой тип: void

Тип `void` — тип с пустым множеством значений.

Главное применение — сообщить о том, что выражение ничего не возвращает (результата нет).

3 Переменные

3.1 Модель памяти

Память компьютера имеет довольно сложное техническое устройство. Кроме того, методы управления памятью могут сильно отличаться в зависимости от используемой архитектуры и ОС.

Модели памяти предоставляют удобную абстракцию для работы, которая в большинстве ситуаций позволяет писать код, не задумываясь о низкоуровневых деталях реализации взаимодействий.

При написании кода мы руководствуемся предоставленной моделью памяти, а реализация и исполнение конкретных инструкций лежит на плечах компилятора/ интерпретатора.

3.2 Модель памяти C++

- Модель памяти C++ представляет из себя *последовательность нумерованных ячеек памяти*, которые называются *байтами*.
- Номер ячейки памяти называется *адресом*.
- Таким образом, *байт* — минимальная адресуемая единица памяти.
- Стандарт C++ гарантирует, что размер байта в точности совпадает с размером объектов типа *char*.
- Все ячейки памяти равноправны. За исключением нулевой ячейки: туда ничего нельзя записать и оттуда ничего нельзя прочитать.
- Последовательности ячеек могут иметь различную интерпретацию в зависимости от того, элемент какого типа мы рассматриваем в данный момент.

3.3 Переменные

Переменная — именованная область памяти. Чтобы создать переменную, необходимо указать ее тип, дать ей имя и, возможно, начальное значение:

```
int x;
short y = 1; // 1 is cast to short
float z(1.5); // 1.5 is cast to float
char t{'@'};
```

Определение переменной выделяет область памяти достаточную для хранения объектов указанного типа и закрепляет за этой областью обозначенное имя.

Однотипные переменные можно создавать в одну строку *[плохой стиль]*.

```
int x = 1, y, z = 3;
```

Переменные можно использовать в выражениях, их значения будут использованы для вычислений:

```
int main() {
  int x = 1;
  int y = 2;
  std::cout << x + 3 * y << '\n'; return 0;
}
```

Как и любую другую сущность, переменную нельзя использовать до того, как она была объявлена:

```
int main() {
  std::cout << x + 3 * y << '\n';
  int x = 1;
  int y = 2;
  return 0;
}
```

```
main.cpp: In function   int   main()   :
main.cpp:4:16: error:    x    was not declared in this scope
  4 | std::cout << x + 3 * y << '\n';
    |              ^
main.cpp:4:24: error:    y    was not declared in this scope
  4 | std::cout << x + 3 * y << '\n';
    |                      ^
```

3.3.1 Автоматическая область действия

Блоки бывают полезны для структурирования кода функции на логические части, а также при использовании с другими операторами (об этом позже в курсе).

Важное свойство блоков состоит в том, что они завершают действие переменных, объявленных внутри блока.

```
int main() {
  int x = 1;
  {
    int y = 2;
    std::cout << x + 3 * y << '\n';
  } // <-- at this point 'y' no longer exists
  return y; // error: there is no 'y' in this block
}
```

Таким образом, получаем, что обычная переменная, объявленная внутри функции, действует с места объявления до *ближайшего* конца блока, внутри которого она объявлена.

3.3.2 Глобальная область действия

Переменные можно объявлять и вне функций. Такие переменные называются *глобальными*.

```
int x = 1;
int main() {
  int y = 2;
  std::cout << x + 3 * y << '\n';
  return 0;
}
```

Глобальные переменные существуют на протяжении всего времени работы программы, но могут быть использованы только после объявления!

Чтобы использовать переменные, объявленные после функции, необходимо объявить переменную `x` внутри функции и указать, что эта переменная определена в другом месте. За это отвечает ключевое слово `extern` :

```
int main() {
  int y = 2;
  extern int x; // declaration without definition
  std::cout << x + 3 * y << '\n';
  return 0;
}
int x = 1; // definition x
```

3.3.3 Классы памяти

В C++ каждый объект в памяти принадлежит одному из 4х классов:

- Автоматическая (стековая) память. Память, в которой хранятся обычные локальные переменные функции. Живут до конца своего блока.
- Глобальная (статическая) память. Память, которая выделяется в начале работы программы (до старта main) и освобождается при завершении.
- Динамическая память (куча). Память выделяется и освобожается вручную. Обсудим позже.
- Потоковая память. Связана с потоком исполнения. Выделяется в начале выполнения потока, освобождается при завершении. В курсе не обсуждается.

3.3.4 Область видимости (scope)

Область видимости (scope) переменной — часть кода, из которой возможен обычный доступ к переменной по ее имени.

Очевидно, что область видимости не превышает области действия переменной.

При этом *scope* может быть меньше области действия:

```
int x = 0;
int main() {
  int x = 1;
  std::cout << x << '\n'; // 1
  return 0;
}
```

В этом примере, область действия обоих `x` распространяется на функцию `main`. Однако внешнее имя скрыто локальным именем.

Для того, чтобы получить доступ к глобальной переменной, можно воспользоваться операцией разрешения области видимости (`::`)

```
int x = 0;
int main() {
  int x = 1;
  std::cout << ::x << '\n'; // 1
  return 0;
}
```

`::` перед именем переменной говорит компилятору о том, что имя нужно искать в глобальной области.

Однако в случае:

```
int x = 0;
int main() {
  int x = 1;
  {
    int x = 2;
    std::cout << x << '\n'; // 2
    std::cout << ::x << '\n'; // 0
  }
  return 0;
}
```

Способа обратиться к промежуточному `x` по его имени нет.

3.3.5 Инициализация переменных

Переменные при определении могут быть инициализированы.

```
int a; // uninitialized variable
int b = 1;
int c(2);
int d{3};
int e{}; // empty curly braces = padding with zeros
```

Последние 4 строки так или иначе инициализируют значение переменной (делают запись в соответствующую область памяти).

Чтобы изменить значение переменной, можно воспользоваться операцией присваивания:

```
int x;
x = 10; // changed the value to 10
int y = 1;
y = 2; // changed the value to 2
```

Важно: инициализация с помощью `=` **НЕ** является присваиванием.

Чтение из неинициализированной переменной приводит к UB. Пользоваться ей можно только после установки конкретного значения.

3.3.6 Потоковое чтение данных

Для чтения значений из потока можно воспользоваться библиотекой `<iostream>` :

```
#include <iostream>
int main() {
  int x;
  int y;
  std::cin >> x >> y;
  std::cout << x + y << '\n';
  return 0;
}
```

Данная программа получает с консоли 2 целых числа и выводит их сумму.

3.3.7 Квалификатор const

Константный объект — объект, к которому применимы только те операции, которые не меняют его логического состояния.

Чтобы объявить константную переменную, необходимо к названию типа добавить слово `const` :

```
const int x = 0; // More preferred option
int const y = 0;
```

Теперь попытка изменения `x` будет приводить к ошибке компиляции:

```
x = 1; // CE
std::cin >> y; // CE
```

3.3.8 Квалификатор volatile

К типу переменной можно дописать ключевое слово `volatile` . Это означает, что операции с этой переменной не должны быть оптимизированы (удалены, переставлены местами с другой операцией и т.п.).

Это бывает полезно, когда доступ к переменной осуществляется не на прямую, а опосредовано.

```
volatile int x = 0;
int volatile y = 0;
```

3.4 Больше о выражениях

3.4.1 Присваивающие операции

Для изменения значения переменной используется присваивание (`=`):

```
y = 5;
x = y;
x = y * y;
```

Помимо обычного присваивания, существуют присваивающие арифметические операции:

```
x += 5; // x = x + 5
y %= 2; // y = y % 2
y |= 9; // z = z | 9
```

Присваивающие операции не только изменяют значение, но и, как и другие, возвращают результат своего исполнения — новое значение.

```
std::cout << (y = 5);      // 5
std::cout << (x = y);      // 5
std::cout << (x = y * y); // 25
```

```
std::cout << (x += 5); // 30
std::cout << (y %= 2); // 1
std::cout << (y |= 9); // 9
```

Это дает возможность строить цепочки присваиваний:

```
x = y = 5;
z = x = y *= y;
```

Более того, оказывается, что результатом присваивающих операций является не просто значение, а как бы сама исходная переменная.

То есть, в результат можно снова что-то присвоить!

```
(x = y) = 5;  // x == 5
(x = y) *= 0; // x == 0
```

При этом заметим, что в качестве левого операнда присваивания не может стоять литерал или результат неприсваивающей операции:

```
5 = x;
x * y = 10;
```

Сборка падает со странной ошибкой:

```
main.cpp:8:3: error: lvalue required as left operand of assignment
    8 | 5 = x;
      | ^
main.cpp:9:5: error: lvalue required as left operand of assignment
    9 | x * y = 10;
      | ~~^~~
```

3.4.2 Резлультат выражения

Результат работы выражения проявляется в

1. Возвращаемом значении
2. Побочных эффектах

Каждое выражение характеризуется типом **возвращаемого значения** и **категорией значения**.

- С типом возвращаемого значения все ясно (если нет, то вернитесь к разделу с фундаментальными типами)
- Категория значения отвечает на вопрос: материален ли результат выражения, то есть, существует ли он в виде объекта в памяти.

```
int x, y =0; // type | side effect | value category
// ---------------------------------------------------
x + y;       // int    | no           | not material
x = 5;       // int    | x stores 5   | material (x)
x *= 2;      // int    | x stores 10  | material (x)
y + (x *=6); // int    | x stores 60  | not material
x / 2.;      // double | no           | not material
```

3.4.3 Категории значений

Материальная категория значений называется *lvalue*, нематериальная — *prvalue*.

Так как *lvalue* значения материальны (соответствуют некоторому расположению в памяти), они могут быть использованы в качестве левого операнда в операцииприсваивания (за некоторыми исключениями).

Теперь природа ошибок становится ясной:

```
main.cpp:8:3: error: lvalue required as left operand of assignment
    8 | 5 = x;
      | ^
main.cpp:9:5: error: lvalue required as left operand of assignment
    9 | x * y = 10;
      | ~~^~~
```

Примечание: помимо *lvalue* и *prvalue* встречается *xvalue*, но о нем будет говорить позже.

3.4.4 Инкремент и декремент

Инкремент/декремент — унарные операции, осуществляющие увеличение/уменьшение аргумента на 1.

Как и присваивание, могут быть применены только к *lvalue*.

```
int x = 0;
x++; // postfix increment (x = 1)
++x; // prefix increment (x = 2)
x--; // postfix decrement (x = 1)
--x; // prefix decrement (x = 0)
```

Если же мы попробуем сделать инкеремент/декремент не *lvalue*, например:

```
int x = 0;
++5;
(x + 1)--;
```

Мы получим следующие ошибки:

```
main.cpp:3:5: error: lvalue required as increment operand
    3 | ++5;
main.cpp:4:6: error: lvalue required as decrement operand
    4 | (x + 1)--;
```

Пост и пре инкременты/декременты отличаются друг от друга возвращаемым значением.

Префиксные операции `++` и `--` возвращают обновленное *lvalue* значение (то есть сам аргумент).

Постфиксные версии возвращают старое значение (до обновления) *rvalue*.

```
int x = 0;
std::cout << ++x << '\n'; // 1 (x == 1)
std::cout << x++ << '\n'; // 1 (x == 2)
--x = 10; // x == 10
x-- = 11; // CE
```

3.4.5 Операция(operator) sizeof

`sizeof` — унарная операция(operator), возвращающая размер объекта в байтах.

- Если `sizeof` применяется к *типу*, то возвращает, сколько памяти в байтах занимают элементы этого типа

```
sizeof(char) == 1; // always
sizeof(int) == 4;  // probably
```

- Если `sizeof` применяется к выражению, то вычисляет размер типа возвращаемого значения.

  Важно: `sizeof` не вычисляет переданное выражение, а лишь анализирует.

```
long long x;
sizeof('a'); // 1
sizeof(1 / 0); // 4
sizeof(x = 1); // 8 (x does not change!)
sizeof(++x); // 8 (x does not change!)
```

3.4.6 Операция(operator) static_cast

`static_cast` принудительно осуществляет допустимые неявные преобразования значений одного типа в другой:

`static_cast<new type> (expression)`

```
int x = 2;
int y = 2000000000;
std::cout << x / y << '\n'; // integer division
std::cout << static_cast<float>(x) / y << '\n';

std::cout << x * y << '\n'; // overflow (UB)
std::cout << static_cast<int64_t>(x) * y << '\n';
```

3.4.7 Операция(operator) ,

операция (operator) "запятая"позволяет объединить несколько выражений в одно:

```
++x, y--, std::cout << x + y, 5;
```

- операция(operator) `,` гарантирует, что сначала будет вычислено выражение слева.
- Результатом операции `,` является результат второго операнда (правый операнд).

3.5 Ключевое слово auto

- До C++11 `auto` использовалось для обозначения переменной в автоматической области памяти (локальная переменная).
- Возможность оказалась настолько неактуальной, что в стандарте C++11 кардинально поменяли смысл слова `auto` (довольно исключительная ситуация).
- Прежний смысл отдали ключевому слову `register` .
- Однако в C++17 отказались и от него, и теперь `register` считается устаревшим (слово зарезервировано для дальнейшего использования).

В наше время `auto` используется для автоматического вывода типа переменной при инициализации

```
int main() {
  auto x = 0; // int
  auto y = 0.0; // double
  auto px = &x; // int*

  auto z; // CE: variable type cannot be determined
}
```

Очень удобно, когда речь идет о длинных именах типов.

Правила вывода типа `auto` совпадают с правилами вывода для шаблонных параметров (за некоторым исключением).

```
int x = 0; const int cx = 1; int& rx = x; int arr[10];

auto y = x; // int
auto cy = cx; // int
auto ry = rx; // int
auto arr_y = arr; // int*

auto& z = x; // int&
auto& cz = cx; // const int&
auto& rz = rx; // int&
auto& arr_z = arr; // int(&)[10]

//Exception
auto x = {1, 2, 3}; // std::initializer_list<int>
```

4 Условная операция(operator) и условный оператор(statment)

4.1 Условная операция(operator)

В языке C++ существует единственная тернарная операция(operator) (принимающая 3 аргумента) — условная операция(operator) (`?:`). Имеет вид `<bool-expr> ? <expr1> : <expr2>` , где:

1. `<bool-expr>` — выражение со значением конвертируемым в bool
2. `<expr1>` , `<expr2>` — выражения с "совместимыми"возвращаемыми значениями.

Если `<bool-expr>` возвращает `true` (или то, что приводится к true), то выполняется `<expr1>` , иначе — `<expr2>` .

Решим с помощью условной операции задачу:

На вход поступают неотрицательные целые числа x и y, найти результат целочисленного деления x на y, если $y \neq 0$, а иначе вывести 1.

```
#include <iostream>

int main() {
  int x;
  int y;
  std::cout << (y != 0 ? x / y : -1) << '\n';
  return 0;
}
```

Проблем с UB нет, так как `x/y` выполняется **только** при `y != 0` .

Попробуем аналогично решить другую задачу:

На вход поступают неотрицательные целые числа x и y, найти результат целочисленного деления x на y, если $y \neq 0$, а иначе вывести "Error".

```cpp
#include <iostream>

int main() {
  int x;
  int y;
  std::cout << (y != 0 ? x / y : "Error") << '\n';
  return 0;
}
```

Получим сообщение с ошибкой:

```
main.cpp:6:24: error: operands to '?:'
have different types 'int' and 'const␣char*'
6 | std::cout << (y != 0 ? x / y : "Error") << '\n';
```

Проблема заключается в том, что типы `int` и "строка"несовместимы, то есть не существует неявного преобразования одного в другое.

4.1.1 Возвращаемый тип

```
<bool-expr> ? <expr1> : <expr2>
```

Коротко, результатом условной операции является наибольший тип, способный вместить результат `<expr1>` и `<expr2>` .

Результирующий тип, разумеется, не зависит от истинности `<bool-expr>` , так как разрешение типов происходит на этапе компиляции, а не во время выполнения:

```cpp
x > 0 ? 1 : 1.0; // return type - double
true ? 0 : 5ll; // return type - long long
false ? "str" : 0; // error - "string" and int incompatible
```

4.1.2 Категория значения

Правила выбора категории значения логической операции:

- Если типы и категории значений обоих выражений совпадают, то выбора нет.
- Если категории значения разные, то результат — prvalue
- Если категории совпадают, а типы отличаются только наличием дополнительного cv-квалификатора, то результат — общая категория + тип с дополнительным cv-квалификатором.
- Если типы разные (даже с точностью до cv), то результат — наиболее подходящий общий тип категории prvalue.

4.2 Условный оператор

Условный оператор(statment) в C++ имеет вид:

```
if ([init] <condition>) <statement-true> [else <statement-false>]
```
,

где:

- `condition` — либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool` .
- `statement-true` — оператор, который выполняется, если `condition` — `true`
- `statement-false` — оператор, который выполняется, если `condition` — `false`
- `init` — либо выражение, либо объявление. Область действия объявленной сущности совпадает с областью условного оператора.

4.2.1 Примеры

Примерами простого использования `if` являются:

```
if (x > 0) std::cout << x << '\n';
```

```
if (int x = 0) { // false
  int y;
  std::cin >> y;
  std::cout << y / x << '\n';
} // here x and y are no longer valid
```

```
if (x != 0 && y / x > 5) return y / x - 5;
```

```
if (x);
```

Напоминание: в `condition` новую переменную инициализировать обязательно:

```
if (int x) std::cin >> x; // error: x = ?
```

4.2.2 Примеры (init)

Примерами использования `if` с инициализацией являются:

```
if (const int x = y + z; x != 5) std::cout << x << '\n';
```

```
if (int x; x = 0) { // false (in init initialization is optional)
  int y;
  std::cin >> y;
  std::cout << y / x << '\n';
}// here x and y are no longer valid
```

```
// init can contain any expression (not just a declaration)
if (std::cin >> x; x != 0 && y / x > 5) return y / x - 5;
```

4.2.3 Примеры (else)

Примерами использования `if` с `else` являются:

```
if (x != 0) std::cout << y / x << '\n';
else std::cout << "Error\n";
```

```
if (x >= 0 && y >= 0) std::cout << "Positives\n";
else {
  int z = x * y;
  std::cout << (z > 0 ? "Negatives\n" : "Different\n");
}
```

```
if (x >= y) {
  if (x >= z) {
    std::cout << x << '\n';
  } else {
    std::cout << z << '\n';
  }
} else if (y >= z) std::cout << y << '\n';
else std::cout << z << '\n';
```

`else` относится к ближайшему неспаренному `if` !

4.3 Оператор(statment) switch

4.3.1 Определение

оператор(statment) switch представляет собой оператор(statment) выбора нужной ветви в зависимости от значений выражения или определения. Он выглядит следующим образом:

`switch ([init] <condition>) statement`

- `init` — либо выражение, либо объявление. Область действия объявленнойсущности совпадает с областью оператора.
- `condition` — выражение или определение, имеющее целый илиперечислимый тип.
- `statement` — произвольный оператор.

Как правило, в качестве `statement` выступает составной оператор, в котором присутствуют метки `case` , `default` и операторы `break` :

```
switch (x) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n"; break;
  default: std::cout << "Other\n";
}
```

Конструкция `case` и `default` выглядит следующим образом:

`case <const-expr>: <statements>`

`default: <statements>`

- `const-expr` — выражение с целым или перечислимым значением, которое вычислимо **на этапе компиляции**.
- `statements` — последовательность операторов.

После вычисления `switch` условия ищется соответствующая метка и управление передается ее первому оператору.

Если нужного значения найдено не было, то осуществляется переход к `default` .

Важно!

После перехода к метке выполняются все операторы, расположенные после нее (даже те, которые лежат под другими метками).

Чтобы завершить выполнение операторов, необходимо написать оператор(statment) `break;`

4.3.2 Примеры без инициализации

Примерами использования `switch` без инициализации являются:

```
int x = ...;
switch (x) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n";
  //after "One" will display "Default"
  default: std::cout << "Other\n";
}
else std::cout << "Error\n";
```

```
int x = 0;
const y = 2;
switch (x * y) {
  case 2 * 2: ... // OK
  case y: ...     // OK
  case x * 2: ... // CE (not evaluated at compile time)
}
```

Напоминание: в качестве условия может стоять только целое число:

```
double x = ...;
switch (x) { //CE
  ...
}
```

4.3.3 Примеры с инициализации

Примерами использования `switch` с инициализацией являются:

```
switch (int x = ...) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n"; break;
  default: std::cout << "Other\n";
}
```

```
switch (std::cin >> x; x * x) {
  case 0:
  case 1: std::cout << "x␣*␣x␣<=␣1\n"; break;
  default: std::cout << "x␣*␣x␣>=␣4\n";
}
```

4.3.4 Атрибут [[fallthrough]]

Зачастую отсутствие `break` — скорее ошибка разработчика, нежели желаемое поведение. Существует не так много ситуаций, когда мы хотим выполнения сразу нескольких подряд идущих веток.

Поэтому компиляторы заботливо выдают предупреждения (ошибку при флаге *-Werror*), если встречают `case` без `break` :

```
main.cpp:10:26: error: this statement may fall through [-Werror=implicit-fallthrough=]
```

Чтобы успокоить компилятор и сказать, что мы так и хотели, можно дописать *атрибут* `[[fallthrough]]`

```
switch (std::cin >> x; x * x) {
  case 0: [[fallthrough]]
  case 1: std::cout << "x␣*␣x␣<=␣1\n"; break;
  default: std::cout << "x␣*␣x␣>=␣4\n";
}
```

Теперь компилятор будет уверен, что вы знаете, что делаете в `case 0` .

5 Циклы

5.1 Цикл while

Цикл - оператор, позволяющий организовать повторяющееся выполнение другого оператора.
`while (<condition>) <statement>`

- `condition` — либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool`
- `statment` — оператор(statment) (может быть составной)

5.1.1 Решение простой задачи

Вводится n целых чисел, найти их сумму.

```
#include <iostream>

int main() {
  int n;
  std::cin >> n;
  int sum = 0;
  while (n > 0) {
    int x;
    std::cin >> x;
    sum += x;
    --n;
    }
  std::cout << sum << '\n';
  return 0;
}
```

5.1.2 Примеры

Сами простыми примерам являются:

```
// endless cycle
while (true) std::cout << 0;
```

```
x = -5;
while (int sqr = x * x) {
  ++x;
  std::cout << sqr << '\n';
}
```

```
// empty loop (spins while x is true)
while (x);
```

Замечание: последний цикл - *Undefined Behaviour*, если x не изменяет своего результата и не имеет побочных действий.

5.2 Цикл do-while

Цикл `do-while` аналогичен циклу `while` , за исключением того, что оператор(statment) цикла выполняется до проверки условия.

```
do <statement> while (<condition>);
```

Таким образом, гарантируется, что цикл совершит хотя бы одну итерацию.

```
int x;
do {
  std::cin >> x;
  std::cout << x * x << '\n';
} while (x);
```

5.3 Цикл for

Цикл `for` имеет следующую структуру в C++:

```
for ([init]; [condition]; [expression]) <statement>
```

- `init` — либо выражение, либо объявление. Область действия объявленной сущности совпадает с областью оператора.
- `condition` — либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool` .
- `expression` — произвольное выражение, выполняющееся в конце итерации
- `statment` — оператор, выполняющийся в цикле

5.3.1 Эквивалентность цикла for и while

Цикл `for`

```
for ([init]; [condition]; [expression]) <statement>
```

эквивалентен циклу `while` слудющего вида:

```
{
  [init];
  while ([condition]) { // while(true), if condition is empty
    <statement>
    [expression];
  }
}
```

но при этом гораздо лучше читаем, поэтому на практике чаще используется `for` .

5.3.2 Примеры

Простыми примерами применениями цикла `for` являются:

```
for (int i = 0; i < n; ++i) std::cout << i << '\n';
```

```
for (int i = 0; i < n; i += 2) {
  std::cout << i << '\n';
}
```

```
for (std::cin >> x; x != 0; std::cin >> x) {
  std::cout << x * x << '\n';
}
```

```
// endless cycle
for (;;) ...
```

```
// analogue of while
for (; x;) ...
```

```
for (int i = 0, j = 0; i < n && j < m; ++i, ++j) ...
```

5.4 Управляющие операторы

5.4.1 Оператор(statement) break

оператор(statement) `break` позволяет досрочно завершить **выполнение цикла**:

```
for(int i =0; i < n; ++i) {
  int x;
  std::cin >> x;
  if (x == 0) {
    std::cout << "Division␣by␣zero\n";
    break;
  }
  std::cout << y / x << '\n';
}
```

5.4.2 Оператор(statement) continue

Оператор(statement) `continue` позволяет досрочно завершить **текущую итерацию**:

```
for(int i =0; i < n; ++i) {
  int x;
  std::cin >> x;
  if (x == 0) {
    std::cout << "Division␣by␣zero\n";
    continue;
  }
  std::cout << y / x << '\n';
}
```

5.4.3 Оператор(statement) безусловного перехода

Оператор(statement) `goto` позволяет совершить "прыжок"в произвольное место функции, обозначенное некоторой "меткой".

```
// the program counts x and exits
int main() {
  int x;
  std::cin >> x;
  goto label;

  int y;
  std::cin >> y;
  std::cout << x + y << '\n';
  label:
  return 0;
}
```

Через `goto` может быть реализован цикл:

```
for (int i = 0; i < n; ++i) ...
// <=>
  int i = 0;
loop:
  if (i < n) {
    ...
    ++i;
    goto loop;
  }
```

Также оператор `goto` лежит в основе оператора `switch`, так как все случаи (`case` и `default`) являются метками для перехода. Поэтому даже следующая программа скомпилируется:

```
switch (x) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n"; break;
  deault: std::cout << "Other\n"; // with a missing letter f
}
```

Оператор `goto` сильно усложняет чтение и отладку программ.
Во всех (даже безвыходных) ситуациях можно обойтись без него.

6 Указатели

6.1 операция(operator) взятия адреса

Согласно модели памяти C++ каждая ячейка имеет свой адрес. Для того, чтобы узнать адрес, по которому лежит объект в памяти, необходимо использовать `&`

```
int x;
std::cout << &x << '\n'; // displayed in hexadecimal
```

Операцию можно применять только к *lvalue* значениям. Более того, **возможность взятия адреса можно использовать в качестве критерия lvalue**:

```
// it is possible
&x, &(x = 8), &(++x);

// that's not possible
&5, &(x + 8), &(x++);
```

Адреса в C++ имеют специальный тип — указатель на тип объекта, у которого был взят адрес.

```
int x;
const float y = 0;

&x; // pointer to int [int*]
&y; // pointer to const float [const float*]
```

6.2 Указатели

Указатель — тип данных, позволяющий хранить адрес другого объекта в памяти.

Пусть `T` — некоторый тип, тогда `T*` — указатель на `T`. Таким образом, указатели — это целое семейство типов.

```
int* // pointer to int
float* // pointer to float
long long* // pointer to long long
const char** // pointer to pointer to const char
double*** // pointer to pointer to pointer to double
```

6.3 Примеры

Примерами использования указателей могут быть следующие:

```
int x = 0;
const float y = 0;

int* px = &x;
int* px2 = &(++x); // == px

const float* cpy = &y;
const int* cpx = &x;

int** ppx = &px;
const int** cppx = &cpx;
```

Указатели учитывают константность!

```
float* py = &y; // CE: py breaks the constness of y
```

6.4 Разыменование

По указателю можно получать значение объекта и даже изменять его.

Для этого воспользуемся операцией разыменования (`*`)

Формально: операция(operator) разыменования — унарная операция(operator), применяемая к указателям и возвращающая *lvalue* низлежащего типа.

```
int x = 1;
float y = 2.5;

int* px = &x;
int* py = &y;

*px = 11;
*py = 3.5;
std::cout << *px << ' ' << *py; // 11 3.5
std::cout << x << ' ' << y; // 11 3.5
```

6.5 Арифметика указателей

Как известно, адреса в C++ представляют собой целые числа. Но арифметика с этими числами представляет собой немного другое, чем работа с `int` , например.

Правила арифметики звучат так:

- Указатели одинакового низлежащего типа можно вычитать друг из друга. Результат — количество элементов данного типа, которое поместится в этом промежутке (разница в байтах / `sizeof(T)`).

```
int x, y, z;
std::cout << &z - &x << '\n'; // most likely 2
```

- К указателям можно прибавлять целые числа. Результат — указатель, сдвинутый на `n` шагов размера `sizeof(T)` .

```
int32_t* p = ...;
p + 5; // + 20 bytes
p - 4; // - 16 bytes
p += 10;
p -= 12;
++p;
p--;
```

- Остальные арифметические операции недопустимы.

Отметим, что разность указателей на переменные или сдвиг указателя на переменную с точки зрения стандарта не дает разумного результата.

Разыменование подобных указателей приводит к *UB*.

Для чего на самом деле нужна арифметика указателей, узнаем, когда будем говорить о массивах.

6.6 Указатели и const

Указатели могут быть константными с двух точек зрения.

- *Указатель на константу.* Константным является объект, на который указывают:

```
int x = 0;
const int y = 0;

const int* px = &x; // or int const*
const int* py = &y; // or int const*

std::cin >> *px; // CE: the object is considered immutable
std::cout << *py; // OK: object is readable
```

- *Константный указатель.* Константным является сам указатель:

```
int x;
int y;

int* const px = &x;

std::cin >> *px; // OK
std::cout << *py; // OK

px = &y; // CE: pointer immutable
++px; // CE
```

Константности можно комбинировать:

```
const int* px = ...; // pointer to constant
int const* py = ...; // pointer to constant
int* const pz = ...; // constant pointer
const int* const pa = ....; // constant pointer to constant
int const* const pa = ...; // constant pointer to constant
```

Лайфхак: читайте типы указателей справа налево.
`int const* const` — constant pointer to constant int

6.7 Указатели на указатели

Указатель — это тип, а значит, на него тоже можно создать указатель.

```
int x;
int* px = &x;
int** ppx = &px;
int*** pppx = &ppx;
```

Указатели на указатели тоже дружат с `const` :

```
const float* const** const p = ...;
// constant pointer to pointer to constant pointer to constant float
```

6.8 Нулевой указатель

Попытка чтения неинициализированной переменной приводит к *UB*. То же самое касается и указателей.

```
int* p;
std::cout << p << '␣' << *p; // UB
```

При этом указателям в общем случае нельзя присвоить конкретного начального числового значения:

```
int* p = &x; // Ok
int* q = p; // Ok
int* r = 10; // CE: conversion from int to int* is forbidden
```

Для того, чтобы задать начальное значение "пустого"указателя и отличить корректный указатель от некорректного, вводится понятие *нулевого указателя*.

Любому указателю можно присвоить значение 0, которое соответствует нулевому адресу

Напоминание: по стандарту по нулевому адресу не может находиться ничего, поэтому разыменование подобного указателя приводит к UB.

```
int* p = 0;
const float* pp = 0;
```

К сожалению, подобная возможность иногда приводит к проблемам, так как возникает ассиметрия: целые числа присваивать нельзя, а 0, внезапно, можно.

Правильным с точки зрения современного C++ способом присвоить нулевое значение указателю является литерал `nullptr` .

```
int* p = nullptr;
const float* pp = nullptr;
```

`nullptr` значение специального типа (`std::nullptr_t`), которое неявно приводится к нулевому указателю любого типа.

Правило: для обозначения нулевых указателей используйте только `nullptr` .

6.9 Указатель на void

Напоминание: `void` — специальный тип, обозначающий отсутствие объекта.

Если хочется сохранить адрес ячейки памяти без учета типа объекта, который там может лежать, можно воспользоваться `void*` :

```
int x;
void* p = &x;

std::cout << p; // Ok
std::cin >> *p; // CE: not sure what to write
std::cout << *p; // CE: not sure what to read
```

6.10 "Висячий"указатель

"Висячим"указателем называется указатель, который указывает на объект, область действия которого уже закончилась.

```
int main() {
  int* p = nullptr;
  {
    int x = 0;
    p = &x;
  }
  return *p;
}
```

Чтение или запись данных по такому указателю приводит к *UB*!

Важно: не допускайте провисания указателей!

6.11 Динамическое выделение памяти

До сих пор мы лишь заводили переменные в глобальной области и на стеке.

У хранения данных в глобальной области и на стеке есть ряд ограничений:

1. Они имеют строго фиксированное время жизни. То есть глобальные переменные будут жить до конца программы, а локальные - до конца текущего блока.
2. Как правило, эти области ограничены несколькими мегабайтами. Однако реальные программы часто требуют гораздо больше.

6.11.1 операция(operator) new

Выражения вида `new T` / `new T(...)` / `..new T`. создают в динамической области объект типа T и возвращают указатель на него.

```
int* pa = new int; // without initialization
int* pb = new int(11); // initialization number 11
int* pc = new int{13}; // initialization number 13
```

Полученный указатель используется для доступа к объекту

Объекты в динамической области живут с момента вызова `new` и до окончания работы программы.

Таким образом, можно создавать объекты внутри блоков, которые будут доступны и вне него:

```
int* p = nullptr;
{
  int x = 11;
  p = new int(13);
}
std::cout << x; // CE: x already destroyed
std::cout << *p; // OK
```

Но это ведет к тому, что память постепенно "засоряется".

6.11.2 операция(operator) delete

Память в динамической области можно (и нужно) очистить до завершения программы, как только она перестала быть нужной.

`delete <pointer>` — удаляет объект, возвращает память системе.

```
int* p = new int;
// ...
delete p;
```

Замечание 1: удалять можно только ту память, которая была выделена с помощью `new` . В остальных ситуациях `delete` приводит к *UB*.

Замечание 2: удаление нулевого указателя — корректная операция(operator).

6.11.3 Утечка памяти

При некорректной работе с указателями или несвоевременном очищении динамической памяти может возникать утечка памяти - неконтролируемый рост лишней памяти.

```
int* p = new int;
p = nullptr; // lost pointer to dynamic memory - leak
new float; // memory allocated, pointer not saved
p = new int; // forgot to remove after use - leak
```

Аккуратное использование `new` и указателей и своевременное использование `delete` - залог успешной борьбы с утечками.

6.11.4 Иные методы выделения динамической памяти

В языке C основным способом выделения динамической памяти была функция `malloc`

Ключевые отличия:

1. Возвращает указатель `void*` .
2. Требует явного указания количества байт.
3. Не инициализирует память.

```
void* v = std::malloc(11);
int* p = static_cast<int*>(std::malloc(sizeof(int)));
*p = 0;
```

Для очищения используется функция `free` :

```
std::free(v);
std::free(p);
```

В C++ есть аналог функций `malloc` / `free` — функции `operator new` / `operator delete` :

```
void* v = operator new(11);
int* p = static_cast<int*>(operator new(sizeof(int)));
*p = 0;

operator delete(v);
operator delete(p);
```

Они используются крайне редко и в специфических ситуациях.
Основной способ — операции `new` / `delete` .

7 Ссылки

7.1 Определение

Ссылка - альтернативное имя объекта в памяти. Ссылку можно воспринимать как указатель, который не нужно разыменовывать.

```
int main() {
  int x = 0
  int& rx = x; // rx - alternate name x
  x = 1;
  std::cout << x << '␣' << rx << '\n'; // 1 1
  rx = 2;
  std::cout << x << '␣' << rx << '\n'; // 2 2
  return 0;
}
```

Как следует из определения, ссылка связывается не с переменной, а с областью памяти:

```
int main() {
  int x;
  int array[10];
  int* p = ...;

  int& rx = x; // x reference
  int& ra = array[0]; // reference to the first element of the array
  int& rp = *p; // reference to the memory pointed to by p
  return 0;
}
```

7.2 Правила работы со ссылками

Существуют основные правила работы со ссылками:

1. Ссылки связываются раз и навсегда. Нельзя поменять область памяти, на которую ссылается ссылка.

```
int& rx = x; // rx <=> x
rx = y; // x is written to y, rx is not associated with y
```

2. Из п.1 следует, что ссылки по определению константные.

```
int& const rx = x; //this is not possible (this is assumed automatically)
```

3. Ссылки могут быть связаны только с областью памяти (*lvalue*).

```
int& x = 0; //CE
```

4. Нельзя создать ссылку на `void` .

5. Нельзя создать ссылку на ссылку

6. Ссылка обязана быть проинициализирована (связана) при создании.

```
int& rx; //CE
```

7. Нельзя создавать массивы ссылок.

```
int& array[2]{x, y}; //CE
```

7.3 Пример

```
int x = 0;
int y = 1;

int& rx = x; // rx is associated with x
int& ry; // CE: reference must be bound at initialization
rx = y; // x is written to y, rx still refers to x
ry = y; // CE

int& rz = rx; // rz is associated x
int&& rt = rx; // reference to reference cannot be created

// References can only be associated with a memory area (lvalue)
int& z = 0; // CE
int& t = x + y; // CE
```

7.4 Ссылка на константу

Можно создать ссылку на константу. Такая ссылка предоставляет права только на чтение.

```
int x = 0;
const int cx = 1;

const int& rx = x; // OK
const int& rcx = cx; // OK
int& rx = cx; // CE
```

Исключение: константные ссылки продлевают время жизни временных объектов, то есть могут связываться с *prvalue*.

```
int& x = 0; // CE
const int& cx = 0; // OK
```

Константная ссылка предоставляет "убежище"(место в памяти) для значений, поэтому у них можно брать адрес и ссылаться на них:

```
const int* p = &cx; // OK
const int& rcx = cx; // OK
```

8 Массивы

8.1 Определение

Массив — это последовательность элементов **одного типа**, расположенных **непрерывно в памяти**, к которым имеется **доступ по индексу** через некоторый **уникальный идентификатор**.

`int array[10];`

Важно: выражение, стоящее в квадратных скобках, должно быть положительным константным значением, известным на этапе компиляции!

```
int n;
std::cin >> n;
int array[n]; // According to the C++ Standard, this is not possible.
```

Рассмотрим массив более подробно.

`int array[10];`

- Данный массив хранит 10 **int'ов**
- Доступ к каждому элементу можно получить с помощью имени `array`, через операцию `[]`:

  `array[0], array[2], array[9]; //numbering from 0`
- **Гарантируется**, что все они идут в памяти подряд, без разрывов:

  `&a[i] - &a[j] == i - j;`

8.2 Пример решения простой задачи

Задача: *Вводится 100 чисел. Вывести их в порядке возрастания.*

```
int array[100];
for (int i = 0; i < 100; ++i) {
  std::cin >> array[i];
}
for (int i = 0; i < 100; ++i) {
  int min_idx = 0; // index of minimum
  for (int j = 1; j < 100; ++j) {
    if (array[j] < array[min_idx]) min_idx = j;
  }
  std::cout << array[min_idx] << ' ';
  array[min_idx] = 1000000; // assume values are less than 1000000
}
```

8.3 Операции над массивом

Операции над массивами проще рассмотреть на примерах.

```
int array[10];
array[1]; // index access operation
sizeof(array); // 40: returns the total size in bytes
sizeof(array) / sizeof(int); // 10: amount of elements
&array; // array address ( int(*)[10] )
```

8.4 Инициализация массивов

- Неинициализированный массив

  `int a[10];`

- Инициализация нулями

  `int a[10]();`

  `int b[20]{};`

- Заполнение значениями

  `int a[5]{1, 2, 3}; //1 2 3 0 0 (the rest is filled with zeros)`

  `int b[]{1, 2, 3}; //1 2 3 (size is calculated automatically)`

- Копирование запрещено

  `int b[5] = a; //CE`

8.5 Связь массивов и указателей

В большинстве ситуаций массив автоматически преобразуется в указатель на свой нулевой элемент:

```
int array[10];
std::cout << array; // the address of the null element is displayed
std::cout << array + 5; // fifth element address
```

Верно и обратное — к указателям можно применять `[]`, то есть воспринимать их как массивы:

```
int* p = array;
std::cout << p[2]; // the second element of the array

p += 5;
std::cout << p[2]; // the seventh element of the array
```

Это работает следующим образом:

```
p[2]; // equivalent *(p + 2)
2[p]; // yes, that's the same *(2 + p)
```

Важно понимать, что массив ≠ указатель на первый элемент.

```
int array[10];
int* p = array;

std::cout << sizeof(array) << '␣' << sizeof(p); // 40 8
&array; // has type int(*)[10]
&p; // has type int**
```

Но в большинстве ситуаций массив действительно ведет себя как указатель (неявно приводится к указателю)

Сравнивать массивы с помощью операций `>`, `<`, `==`, ... не имеет смысла, так как реально сравниваются указатели на первые элементы.

```
int a[]{1, 2, 3};
int b[]{1, 2, 3};
a == a; // always true
a == b; // always false
```

8.6 Многомерный массив

Массивы могут иметь несколько размерностей

```
int a[10]; // one-dimensional array
a[1]; // accessing an element
int b[5][20]; // two-dimensional array
b[3][11]; // accessing an element
int c[7][9][3]; // 3D array
c[3][8][0]; // accessing an element
...
```

8.6.1 Инициализация многомерных массивов

Инициализация многомерных массивов может быть произведена различными вариантами:

```
int a[2][3]{1, 2, 3, 4}; // padding by line
// 1 2 3
// 4 0 0

int b[][2]{1, 2, 3, 4}; // the first dimension can be inferred
// 1 2
// 3 4

int c[3][2]{{1, 2}, {3, 4}, {5, 6}}; // 1 2
// 3 4
// 5 6
```

8.6.2 Многомерные массивы и указатели

Как и одномерные массивы, многомерные приводятся к указателю на первый элемент. Первый элемент многомерного массива — массив меньшей размерности.

```
int a[1][2][3];
a[0]; // type int[2][3]

int b[2][3];
b[1][1] <=> *(b[1] + 1) <=> *(*(b + 1) + 1) <=> *(&b[0][0] + 3 + 1);
```

8.7 Правила работы с массивами

1. Нельзя создавать массивы ссылок и массивы функций, но можно создавать массивы указателей и массивы указателей на функции.

```
int& a[10]; // CE
int b[20](int); // CE

int* c[30]; // OK (array of pointers)
int (*d[40])(int); // OK (array of function pointers)
```

2. Нельзя создать массив с неизвестным числом элементов, но можно его объявить.

```
int a[]; // CE (definition)
int b[] = {1, 2, 3}; // OK: int[3]
extern int c[]; // OK (announcement)
```

3. При сравнении массивов сравниваются адреса нулевых элементов (не значения!). Это значит, что результат сравнения на равенство для разных массивов всегда `false` .

```
int a[3]{1, 2, 3};
int b[3]{1, 2, 3};
std::cout << (a == b) << '␣' << (a == a); // 0 1
```

4. Массивы нельзя присваивать друг другу (исключение — строки при инициализации).

```
int a[3];
int b[3]{1, 2, 3};
a = b; // CE
```

5. Лайфхак: если массив — это поле структуры или класса, то чудесным образом присваивание начинает работать

```
struct S {
  int array[3];
};

S c{1, 2, 3};
S d{4, 5, 6};
c = d; // OK (c.array == {4, 5, 6})
```

8.8 Динамические массивы

Создаются с помощью оператора `new[]`, который возвращает указатель на нулевой элемент массива.

`int* array = new int[10];`

Так как результат — указатель на элемент (не на массив!), его нельзя использовать в предыдущих контекстах

```
sizeof(new int[10]) == sizeof(int*); // 8
sizeof(*array) == sizeof(int); // true
```

Размер динамического массива, в отличие от стекового, **не обязан** быть константой времени компиляции!

```
int n = 10;
// Work with n ...
int a[n]{1, 2, 3}; // prohibited by the standard, but compilers allow
int* b = new int[n]{1, 2, 3}; // OK
```

И не забывайте убирать за собой!

Важно: память, выделенную с помощью `new[]`, необходимо очищать с помощью `delete[]`

```
int* array = new int[10];
// ...
delete[] array;
```

8.9 Многомерные динамические массивы

Многомерный динамический массив можно моделировать массивом массивов:

```
// memory allocation
int** array = new int*[n];
for (int i = 0; i < n; ++i) {
  array[i] = new int[m];
}

// filling
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < m; ++j) {
    std::cin >> array[i][j];
  }
}

// cleansing
for (int i = 0; i < n; ++i) {
  delete[] array[i];
}
delete[] array;
```

9 Объявления указателей, ссылок и массивов*

9.1 Общие принципы

Модификатор указателя, ссылки, массива распространяется только на ближайшую переменную:

```
int* a, b, c; // int*, int, int
int x, &y, z; // int, int&, int
int p, q, r[10]; // int, int, int[10]
```

В том числе поэтому многими кодстайлами (в том числе нашим) рекомендуется писать по одному объявлению на строку:

```
int* a;
int b;
int c;
```

9.2 Указатели и ссылки на массивы

Чтобы задать указатель/ссылку на массив необходимо "привязать"модификатор `*` / `&` к имени переменной:

```
int (*a)[10]; // pointer to array
int (&a)[10]; // array reference
```

Это правило распространяется и на объявления "в одну строку":

```
int a, *b[10], (&c)[20];
```

9.3 Сложные комбинации

Чтобы читать сложные типы, состоящие из комбинации ссылок, указателей и массивов, прибегнем к методу улитки:

- Находим имя переменной
- Двигаемся направо до первой закрывающейся скобки
- Двигаемся налево до первой открывающейся скобки
- Читаем подряд все встретившиеся типы и модификаторы

Так, например, если нас просят задать переменную типа "ссылка на массив из 15 указателей на массивы `int` 'ов размера 10 то получим
`int (*(&a)[15])[10];`

10 С-строки

10.1 Строковые литералы

Строковыми литералами называются объекты, которые находятся внутри ". Немного о строковых литералах:

1. Представляют собой массивы символов с 0 на конце
2. Могут быть скопированы при инициализации (исключение из общего правила)
3. Сравнение происходит непоэлементно, а сравниваются адреса первых элементов как с массивами. Но у строковых литералов есть особый способ работы:
    - Массив, с которым связан строковый литерал, лежит в статической (глобальной) области памяти (в таблице строковых литералов).
    - Компилятор, анализируя исходный код, помещает каждый попавшийся строковый литерал в отдельный буфер (отдельная строка таблицы).
    - Как правило, одинаковые литералы ссылаются на одну и ту же область памяти, поэтому их сравнение путем сравнения указателей может давать верный результат (но это не гарантировано стандартом!).

10.2 Строки

- Строка — часть массива элементов char, ограниченная символом 0
- Правила работы со строками такие же, как и с обычными массивами

11 Функции

11.1 Структруное программирование

Структурное программирование — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков (логически объединенных последовательностей инструкций).

Основные принципы структурного программирования:

- Используются операторы последовательности, ветвления и цикла.
- Операторы могут быть вложены друг в друга.
- Повторяющиеся фрагменты объединяются в подпрограммы (**функции**)
- ...

11.2 Определение функции

Функция — элемент программы, который связывает последовательность инструкций с идентификатором (именем функции) и списком параметров

В общем виде определение функции выглядит так:

```
ReturnType Name(Type1 p1, Type2 p2, ...) {
  // function body
}
```

Например:

```
// Max function: returns an int, takes 2 ints
int Max(int x, int y) {
  return x > y ? x : y;
}
```

Далее функция может быть вызвана из какой-нибудь другой функции:

```
int main() {
  int a;
  int b;
  std::cin >> a >> b;
  std::cout << Max(x, y) << '\n';
  return 0;
}
```

Как и любую другую сущность, перед использованием функцию необходимо объявить, но не обязательно определять.

```
int Max(int x, int y); // Function declaration or prototype

int main() {
  int a;
  int b;
  std::cin >> a >> b;
  std::cout << Max(x, y) << '\n';
  return 0;
}

int Max(int x, int y) {  // Function definition
  return x > y ? x : y;
}
```

11.3 Объявление функций

Данный синтаксис называется объявлением (прототипом) функции (без определения).

`int Max(int x, int y)`

Объявление (возможно с определением) должно обязательно предшествовать использованию функции.

При объявлении функции можно опускать имена параметров:

`int Max(int, int )`

11.3.1 ODR (One Definition Rule)

У одной и той же функции может быть несколько объявлений в программе. Но определение обязано быть одно!

```
// OK
int Max(int, int y);
int Max(int a, int b);
int Max(int, int);

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

```
// Error
int Max(int x, int y) {
  return x > y ? x : y;
}

int Max(int a, int b) {
  return a > b ? a : b;
}

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

11.4 Параметры функций

11.4.1 Определения

Параметры, указанные в объявлении функции, называются *формальными параметрами*. Параметры, переданные в функцию — *фактическими параметрами*.

```
int Max(int x, int y) { // x, y --- formal parameters
  return x > y ? x : y;
}
int main() {
  std::cout << Max(1, -1) << '\n'; // 1, -1 - actual parameters
  return 0;
}
```

11.4.2 Копирование параметров

При передаче фактических параметров они копируются в формальные.

```
int Max(int x, int y) { // x, y - local variables of the Max function
  x = x > y ? x : y;
  return x;
}

int main() { // x, y - local variables of the main function
  int x = 0;
  int y = 1;
  std::cout << Max(x, y) << '\n'; // 1
  std::cout << x << ' ' << y << '\n'; // 0 1
  return 0;
}
```

11.4.3 Параметры по умолчанию

Некоторые параметры могут иметь значение по умолчанию. В этом случае они могут быть опущены при вызове функции.

Параметры со значениями по умолчанию могут идти только в конце списка.

```
int Max(int x, int y = 0); // OK
int Min(int x = 1, int y = 0); // OK
int Sum(int x = 0, int y); // CE

int main() {
  Max(1, 2);
  Max(1);
  Min(1, 2);
  Min(1);
  Min();
  return 0;
}
```

Параметры по умолчанию должны быть указаны хотя бы 1 раз, а в дальнейшем могут быть опущены:

```
int Max(int, int = 0); // so it is also possible
int Max(int, int); // declaring the SAME function

int main() {
  ...
  Max(a);
  ...
}

int Max(int x, int y) {
  return x > y ? x : y;
}
```

11.5 Возвращаемое значение

- Тип возвращаемого значения указывается перед именем функции.
- Результат возвращается с помощью оператора `return` .
- Значение выражения, стоящего справа от `return` , при необходимости (и возможности) преобразуется в возвращаемый тип.

```
int F() { return 0; } // OK
int G() { return 1.0; } // OK 1.0 is converted to 1
int H() { return "Hello"; } // CE: string is not converted to int
```

Операторов `return` может быть несколько:

```
int Calculate(int x, int y, char c) {
  if (c == '+') {
    return x + y;
  }
  if (c == '-') {
    return x - y;
  }
  if (c == '*') {
    return x * y;
  }
  return x / y;
}
```

Замечание: `else` после return необязателен (`return` завершает выполнение функции)

Если функция ничего не вернет, то ошибки не будет, но обращение к результату в этом случае приведет к *UB*:

```
int Max(int x, int y) {
  if (x > 0) {
    return x > y ? x : y;
  }
}

int main() {
  Max(-1, -1); // OK
  std::cout << Max(-1, -1); // UB
  return 0;
}
```

Чтобы указать, что функция ничего не возвращает, используется тип `void` :

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

int main() {
  Print(0);
  return 0;
}
```

Для досрочного завершения такой функции используется `return` без параметра.

```
void Print(int x) {
  if (x < 0) return;
  std::cout << x << '\n';
}
```

11.6 Передача указателей, массивов, ссылок

11.6.1 Проблема

Довольно часто (например, при сортировке) возникает необходимость обменять значения двух переменных: Логично написать функцию, которая бы занималась этим:

```
void Swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

Данная функция обменивает лишь локальные копии. Исходные перемнные не меняются:

```
int main() {
  int x = 0;
  int y = 1;
  Swap(x, y); // x == 0, y == 1
  return 0;
}
```

Для решения этой проблемы можно использовать передачу указателей и ссылок. Об этом ниже.

11.6.2 Передача указателей

Чтобы из функции получить доступ к локальным переменным другой функции, можно воспользоваться указателями:

```
void Swap(int* px, int* py) { // accept addresses of ints
  int tmp = *px;
  *px = *py; // change value at px address
  *py = tmp; // change value at py address
}

int main() {
  int x = 0;
  int y = 1;
  Swap(&x, &y); // x == 0, y == 1
  return 0;
}
```

11.6.3 Передача ссылок

Более удобным способом является передача параметров по ссылке:

```
void Swap(int& x, int& y) { // accept references
  int tmp = x;
  x = y; // change the value referred to by x
  y = tmp; // change the value referred to by y
}

int main() {
  int x = 0;
  int y = 1;
  Swap(x, y); // x == 0, y == 1
  return 0;
}
```

11.6.4 Передача по константной ссылке

Чтобы избежать копирования при передаче параметров в функции, можно также воспользоваться передачей по ссылке.

Чтобы гарантировать, что исходная переменная не изменится, можно передать константную ссылку:

```
int Max(const int& x, const int& y) {
  return x > y ? x : y;
}
```

Замечание: копирование примитивных типов — очень дешевая операция(operator), поэтому в таких ситуациях лучше использовать передачу по значению. Передача по ссылке имеет смысл для "тяжелых"объектов (структур, классов)

11.6.5 Передача массивов

Самый простой способ передать массив в функцию — передать указатель на начало + размер массива:

```
int Max(const int* array, int size) {
  int max = array[0]; // assume size > 0
  for (int i = 1; i < size; ++i) {
    if (max < array[i]) max = array[i];
  }
  return max;
}

int main() {
  int array[100]; // or int* array = new int[100];
  for (int i = 0; i < 100; ++i) std::cin >> array[i];
  std::cout << Max(array, 100) << '\n';
  // delete[] array;
  return 0;
}
```

Передача массивов по значению не дает ожидаемого результата.

```
void f(int array[50]); // <=> void f(int* array)

int main() {
  int normal[50];
  f(normal); // OK

  int large[100];
  f(large); // OK, but can't process items beyond 50

  int small[10];
  f(small); // Ok can't access memory beyond 10

  return 0;
}
```

Тип массива в качестве параметра автоматически преобразуется в указатель.
Решением данной задачи является передача массива по указателю:

```
void f(int (*array_ptr)[50]);

int main() {
  int normal[50];
  f(&normal); // OK

  int large[100];
  f(&large); // CE

  int small[10];
  f(&small); // CE

  return 0;
}
```

Или аналогично можно передать по ссылке:

```
void f(int (&array_ref)[50]);

int main() {
  int normal[50];
  f(normal); // OK

  int large[100];
  f(large); // CE

  int small[10];
  f(small); // CE

  return 0;
}
```

11.7 Статические локальные переменные

Статические переменные инициализируются **один раз** и живут до конца программы, то есть сохраняют свои значения между вызовами.

Область действия — глобальная, область видимости — блок, в котором объявлена.

По умолчанию инициализируются нулем.

Пример:

```
int F() {
  static int x = 1;
  return x++;
}

int main() {
  std::cout << F() << '\n'; // 1
  std::cout << F() << '\n'; // 2
  std::cout << F() << '\n'; // 3
}
```

11.8 Возврат массивов, указателей и ссылок

11.8.1 Проблемы

Массивы нельзя не только принимать по значению, но и возвращать из функций:

```
int[10] Function() { // Error
  int array[10]{};
  return array;
}
```

Но даже если мы попробуем вернуть указатель или ссылку, например:

```
int* F() {
  int x = 0;
  return &x;
}

int& G() {
  int x = 0;
  return x;
}
```

То у нас будет проблема, так как возврат указателя или ссылки на локальную переменную функции — опасная операция(operator), так как после завершения работы функции переменная уничтожается. Следовательно, получаем провисший указатель/ссылку.

`std::cout << *F() << '␣'<< G() << 'n'; //UB`

11.8.2 Допустимые варианты

Допустимыми вариантами являются:

- Возврат указателя/ссылки на объект в динамической памяти

```
int* CreateArray(int size) {
  return new int[size];
  // hopefully the caller remembers to call delete[]
}
```

- Возврат указателя/ссылки на объект, полученный с помощью указателя/ссылки

```
int& Get(int* p) {
  return *p;
  // we believe that p is a valid pointer
}
```

- Возврат указателя/ссылки на глобальную переменную

```
int x = 0;

int* GetPointer() {
  return &x;
}
```

- Возврат указателя/ссылки на статическую переменную

```
int& Get() {
  static int x = 0;
  return x;
}
```

11.9 Присвоение возвращаемому значению

Если функция возвращает ссылку, то ее результату можно присваивать (и вообще работать с ее результатом как с переменной):

```
int& F();

int& r = F();
F() = 5;
int* p = &F();
```

Теперь должно быть понятно, как работают lvalue выражения (присваивания, ++ и т.п.) — они возвращают не по значению, а по ссылке!

```
int& operator+=(int& x, int y) {
  x = x + y;
  return x;
}
```

11.10 Перегрузка функций

11.10.1 Наследие от C

В языке C функции не могут иметь одинаковые имена, даже если принимают разные типы и количество параметров. Примеры данного наследия можно найти, например, в библиотеке `cmath` :

```
int abs(int x);
long labs(long x);
long long llabs(long long x);
float fabsf(float x);
double fabs(double x);
long double fabsl(long double x);
```

11.10.2 Современный C++

В C++ разрешено заводить функции с одинаковыми именами, различающиеся списком параметров. Данный механизм называется *перегрузкой функций*:

```
int abs(int x);
long abs(long x);
long long abs(long long x);
float abs(float x);
double abs(double x);
long double abs(long double x);
```

11.10.3 Примеры перегрузки

Для того, чтобы перегрузить функцию необходимо, чтобы параметры функции имели либо **разный тип**:

```
int Max(int x, int y) {
  return x > y ? x : y;
}
int Max(const int* array, int size) {
  int res = array[0];
  for (int i = 0; i < size; ++i) res = Max(res, array[i]);
  return max;
}
```

Либо **разное число параметров**:

```
int Minus(int x) { return -x; }
int Minus(int x, int y) { return x - y; }
```

Важно!

Если параметры одинаковые, а функции различаются лишь возвращаемым типом, то это не является перегрузкой функций и является ошибкой.

```
int Abs(int x) { return x > 0 ? x : -x; }
double Abs(int y) { return x > 0 ? x : -x; }
```

11.10.4 Разрешение перегрузки

Процесс выбора подходящей функции называется *разрешением перегрузки*.

```
int F(int) { return 1; }
int F(double) { return 2; }
int F(char) { return 3; }

F(0);   // 1
F(0.0); // 2
F('0'); // 3
```

Разрешение перегрузки происходит только на основе имени функции и типов переданных аргументов (сигнатура функции).

```
long F(long) { return 1; }
long long F(long long) { return 2; }

long x = F(0); // Error: ambiguous call
```

Общее правило: выбирается наиболее подходящая по параметрам функция, требующая наименьшее число преобразований (conversion) типов.

Замечание: расширение (promotion) типа (int -> long -> long long или float -> double) не является преобразованием.

Замечание 2: float -> long double, double -> long double считаются преобразованиями, а не расширениями.

Более подробные правила выглядят следующим образом:

Шаг №1: C++ пытается найти точное совпадение. Это тот случай, когда фактический аргумент точно соответствует типу параметра одной из перегруженных функций.

Например:

```
void print(char *value);
void print(int value);

print(0); // exact match with print(int)
```

Хотя 0 может технически соответствовать и print(char *) (как нулевой указатель), но он точно соответствует print(int). Таким образом, print(int) является лучшим (точным) совпадением.

Шаг №2: Если точного совпадения не найдено, то C++ пытается найти совпадение путем дальнейшего неявного преобразования типов. Таких преобразований достаточно много. Если вкратце, то:

- char, unsigned char и short конвертируются в int
- unsigned short может конвертироваться в int или unsigned int (в зависимости от размера int)
- float конвертируется в double
- enum конвертируется в int

Например:

```
void print(char *value);
void print(int value);

print('b'); // match with print(int) after implicit conversion
```

Шаг №3: Если неявное преобразование невозможно, то C++ пытается найти соответствие посредством стандартного преобразования. В стандартном преобразовании:

- Любой числовой тип будет соответствовать любому другому числовому типу, включая unsigned (например, int равно float)
- enum соответствует формальному типу числового типа данных (например, enum равно float)
- Ноль соответствует типу указателя и числовому типу (например, 0 как char * или 0 как float)
- Указатель соответствует указателю типа void

Например:

```
struct Employee; // let's miss the definition
void print(float value);
void print(Employee value);

print('b'); // 'b' is converted to match the print(float) version
```

Обратите внимание, все стандартные преобразования считаются равными. Ни одно из них не считается выше остальных по приоритету.

Шаг №4: C++ пытается найти соответствие путем пользовательского преобразования. Хотя мы еще не рассматривали классы, но они могут определять преобразования в другие типы данных, которые могут быть неявно применены к объектам этих классов.

Замечание про const: При выборе перегрузки const на верхнем уровне игнорируется. То есть `F(int)` и `F(const int)` задают одинаковые функции!

```
void F(int); /* <=> */ void F(const int);
void F(int*); /* <=> */ void F(int* const);
void F(const int*); /* <=> */ void F(const int* const);

void F(int&); /* != */ void F(const int&);
void F(int*); /* != */ void F(const int*);
```

11.11 Рекурсия

Рекурсия — метод решения задач, основанный на решении аналогичной задачи меньшего размера.

Рекурсивная функция — функция, которая вызывает сама себя.

11.11.1 Прямая рекурсия

Наиболее простой является *прямая рекурсия* — ситуация, когда функция вызывает себя непосредственно (напрямую).

Пример рекурсивной функции — функция, вычисляющая факториал:

```
int Factorial(int n) {
  return n <= 1 ? 1 : n * Factorial(n - 1);
}
```

Важной частью рекурсивных функций является условие завершения рекурсии.

11.11.2 Рекурсия vs Итерации

Ясно, что эту же задачу (факториал) можно решить итеративно:

```
int Factorial(int n) {
  int res = 1;
  for (int i = 2; i <= n; ++i) {
    res *= i;
  }
  return res;
}
```

Рекурсивный алгоритм более лаконичен и "математичен но требует $\mathcal{O}(n)$ памяти для вычисления (стек рекурсии — хранение локальных переменных, адреса возврата, контекста и т.д.)

Сравним следующие алгоритмы:

```
int Fibonacci(int n) {
  return n <= 1 ? 1 : Fibonacci(n - 2) + Fibonacci(n - 1);
}

int Fibonacci(int n) {
  int a = 0;
  int b = 1; // f(0)
  for (int i = 0; i < n; ++i) {
    b += a; // f(i)
    a = b - a; // f(i - 1)
  }
  return b;
}
```

Рекурсивный алгоритм не только требует больше памяти, но и работает за $\Omega(1.5^n)$
Таким образом при использовании рекурсии важно помнить:

- Любой рекурсивный алгоритм можно свести к итеративному
- Рекурсия, как правило, более лаконична, но требует больше ресурсов.
- Рекурсию имеет смысл использовать, если сложность итеративной реализации окупает прирост в производительности.

11.11.3 Косвенная рекурсия

Косвенно рекурсивной называется функция, которая вызывает себя через вызов другой (других) функции.

При использовании косвенной рекурсии важно помнить, что для использования любой сущности необходимо ее сначала объявить. Поэтому для таких функций вначале пишутся все используемые прототипы.

```
int F(int n);
int G(int n);

int F(int n) {
  return n <= 1 ? n : 1 + G(n);
}

int G(int n) {
  return n <= 1 ? n : n + F(n - 1);
}
```

11.12 Указатели и ссылки на функцию

Во время исполнения программы код функции (инструкции) хранится в памяти, а, следовательно, на функцию можно создать указатель или ссылку:

```
// A function that takes an int and returns an int*
int* F(int);

// Pointer to a function that takes an int and returns an int*
int (*G)(int);
```

Указатель/ссылку на функцию можно проинициализировать адресом или самой функцией:

```
int F(int) { ... }

int (*f_pointer)(int) = &F;
// Functions, like arrays, are reduced to pointers.
int (*f_pointer2)(int) = F;


int (&f_reference)(int) = F;
```

11.12.1 Применение

```
(*f_pointer)(1); // call F(1)
// call F(3) (function pointer does not need to be dereferenced)
f_pointer2(3);

f_reference(5); // call F(5)
```

Как правило, указатели и ссылки на функцию используются для передачи их в другие функции.

Например, если хотим задать сортировку по другому критерию:

```
bool Greater(int x, int y) {
  return x > y;
}

void Sort(int* begin, int* end, bool (*compare)(int, int)) {
  ...
  if (compare(x, y))
  ... ...
}

// ...

Sort(array, array + n, Greater); // non-ascending sort
```

Указатели на функции (в отличие от функций) можно хранить в массивах. Таким образом, можно обращаться к ним в цикле:

```
int (*transform[10])(int) = {Action0, Action1, ..., Action9};

int x;
std::cin >> x;
for (int i = 0; i < 10; ++i) {
  x = transform[i](x);
}
```

11.12.2 Правила чтения

К правилам чтения типов добавляется условие: если при движении слева направо встречаем открывающуюся круглую скобку, то далее идут параметры функции

Например:

- `int (*(*a)[5]) (int, float)` — Указатель на массив из 5 указателей на функции `int(int, float)`
- `int* (*b[10]) (int(*)(int))` — Массив из 10 указателей на функции `int* (int(*)(int))`
- `int (*(*a) ()) (const int&)` — Указатель на функцию без аргументов и возвращающую `int(*)(const int&)`

11.13 Шаблоны функций

Шаблоны функций являются логическим продолжением перегрузки функций. Так, например, функция модуля перегруженная для различных типов данных имеют один и тот же код внутри:

```
int abs(int x) { return x > 0 ? x : -x; }
long abs(long x) { return x > 0 ? x : -x; }
long long abs(long long x) { return x > 0 ? x : -x; }
float abs(float x) { return x > 0 ? x : -x; }
double abs(double x) { return x > 0 ? x : -x; }
long double abs(long double x) { return x > 0 ? x : -x; }
```

Для удобства мы хотели бы это поменять и написать один шаблон:

```
template <class T>
T abs(T x) {
  return x > 0 ? x : -x;
}
```

11.13.1 Синтаксис

- В начале объявляется список шаблонных параметров:

```
template <class T> or template <typename T>
```

- Далее следует определение (или объявление) шаблонной функции.

```
template <class T>
T Abs(T x) {
  return x > 0 ? x : -x;
}

template <class T>
T Sum(T x, T y) {
  return x + y;
}
```

- Допустимо использование нескольких шаблонных параметров.

```
template <class T, class U>
void Print(T x, U y) { std::cout << x << ' ' << y; }
```

11.13.2 Применение шаблонов

После того как шаблон функции объявлен его можно использовать.

Нужный тип выводится автоматически *по переданным аргументам* (при наличии).

```
template <class T>
T Abs(T x) { return x > 0 ? x : -x; }

Abs(0.0); // double Abs(double)

template <class T>
T Sum(T x, T y) { return x + y; }

Sum(1, 1); // Ok [T == int]
Sum(1, 0.0); // CE ([T == int] or [T == double])

template <class T>
T GetZero() { return 0; }

GetZero(); // CE - type cannot be inferred
```

Если результат вывода не устраивает или вывод типа невозможен, можно заставить компилятор вызвать функцию с конкретным типом:

```cpp
template <class T>
T Sum(T x, T y) { return x + y; }

Sum<long>(1, 1); // Ok [T == long]
Sum<double>(1, 0.0); // Ok [T == double]

template <class T>
T GetZero() { return 0; }

GetZero<float>(); // Ok [T == float]
```

11.13.3 Вывод типа шаблона при передачи по значению

При передаче аргумента по значению тип `T` выводится по следующим правилам:

1. CV-квалификаторы (`const` , `volatile`) игнорируются.
2. Ссылки отбрасываются.
3. Массивы низводятся до указателей.
4. Функции низводятся до указателей на функцию
5. Типы соответствующие одному шаблонному типу `T` должны совпадать (после выполнения всех действий выше).

Пример:

```cpp
template <class T>
void f(T x, T y);

int x = 0;

const int cx = 1;
f(x, x); // Ok: [T=int]
f(x, cx); // Ok: [T=int]
f(cx, cx); // Ok: [T=int]

int& rx = x;
int arr[11];

f(cx, rx); // Ok: [T=int]
f(&rx, arr); // Ok: [T=int*]
f(&cx, &x); // CE: [T=const int*] or [T=int*]
f(0, 0.0); // CE: [T=int] or [T=double]
f<double>(0, 0.0); // Ok: [T=double] (type is not inferred, but substit
```

В случае 2 типов шаблона, они также определяются автоматически:

```
template <class T, class U>
void f(T x, U y);

// You can output both parameters (if they are outputable)
f(0, 0.0); // Ok: [T=int, U=double]

// You can explicitly specify both
f<double, double>(0, 0.0); // Ok: [T=double, U=double]

// You can specify the first one, the second one will be displayed auto
f<float>(0, 0.0); // Ok: [T=float, U=double]
```

11.13.4 Вывод типа шаблона при передаче по ссылке

Отличие от передачи по значению в том, что при передаче по ссылке или указателю низведений типов не происходит.

```
template <class T>
void f(T& x) { ... }

int x = 0;
const int cx = 1;
int& rx = x;
int arr[10];

f(x); // Ok: void f<int>(int& x);
f(cx); // Ok: void f<const int>(const int& x);
f(rx); // Ok: void f<int>(int& x); no reference to reference
f(arr); // Ok: void f<int[10]>(int (&x)[10])
f(0); // CE: you cannot create references to temporary values

f<const int>(0); // Ok: void f<const int>(const int&)
// or
template <class T>
void f(const T& x) { ... }
```

11.13.5 Параметры шаблона по умолчанию

Можно указать значение шаблонного параметра по умолчанию, тогда, в случае если тип невозможно вывести, будет использоваться значение по умолчанию.

```
template <class T = int>
T GetZero() { return 0; }

GetZero(); // Ok: [T=int]
GetZero<double>(); // Ok [T=double]
```

Можно ссылаться на предыдущие шаблонные параметры

```
template <class T, class U = T>
U f(T x) { ... }

f<int, double>(0); // Ok: [T=int, U=double]
f<float>(0); // Ok: [T=float, U=float]
f(0); // Ok: [T=int, U=int]
```

Важно! Значения аргументов по умолчанию не могут использоваться для вывода шаблонного типа

```
template <class T>
void f(T x = 0) { ... }

f(); // CE: type T is not deduced!
```

Можно делать только так:

```
template <class T = int>
void f(T x = 0) { ... }

f(); // Ok: [T=int]
```

11.13.6 Инстанцирование шаблонов

Шаблонные функции работают не как обычные функции, для них выполнены следующие правила:

- Шаблон функции - это **НЕ** функция!
- Генерации исполняемого кода не происходит, если шаблон ни разу не вызван.
- В случае вызова шаблонной функции создается лишь нужная версия (с вызываемыми параметрами).
- Процесс генерации кода из шаблона называется *инстанцированием* шаблона.

```
template <class T>
void f(T x) { ... }

int main() {
  f(0); // instantiated f<int>
  f(0.0); // instantiated f<double>
  f(1); // Used by f<int> (already instantiated)
  return 0; // there are no other versions of f<T>!
}
```

Можно явно попросить инстанцировать шаблон (даже если он не используется).

```
template <class T>
void f(T x) { ... }

template void f(float); // Explicitly instantiate f<float>

int main() {
  f(0); // instantiated f<int>
  f(0.0); // instantiated f<double>
  f(1); // Used by f<int> (already instantiated)
  return 0; // There are f<int>, f<double>, f<float>
}
```

Это может быть полезно в случае многофайловых программ, когда вы не хотите инстанцировать одну и ту же функцию несколько раз в разных единицах трансляции.

Компиляция шаблонов происходит в два этапа:

1. При объявлении проверяется лишь синтаксис языка и условия, которые не зависят от параметра шаблона
2. Во время инстанцирования происходит полная проверка кода на корректность и генерация машинного кода

```
template <class T>
void f(T x) {
  x = x + x; // Stage 2 (is it possible to add and assign T?)
  g(); // Stage 1 (does not depend on T)
  g(x); // Stage 2 (depends on T)
  static_assert(sizeof(char) == 1); // 1 stage (independent of T)
  static_assert(sizeof(T) > sizeof(char)); // Stage 2 (depends on T)
}
```

11.13.7 Перегрузка шаблонов функций

Как и обычные функции шаблоны можно перегружать.

Общие правила выбора шаблона:

- Точные соответствия всегда побеждают остальные перегрузки.
- Если есть несколько точных соответствий, выиграет соответствие с меньшим числом подстановок и приведений типов.
- При прочих равных обычная функция предпочтительнее шаблона.

```
template <class T, class U>
int f(T x, U y) { return 1; }

template <class T>
int f(T x, T y) { return 2; }

int f(int x, int y) { return 3; }

f(0, 0.0); // 1 (exact match)
f(0.0, 0.0); // 2 (fewer substitutions)
f(0, 0); // 3 (priority over non-template function)

//If we want to use a template
f<>(0, 0); // 2
```

11.13.8 Специализация шаблона

Представим такую ситуацию:

```
template <class T>
T abs(T x) { return x > 0 ? x : -x; }

struct Complex {
  double re;
  double im;
};

Complex c{3, 4}; // 3 + 4i
abs(c); // CE: no match for operator>
```

Решение - специализация шаблона.

Специализация шаблона позволяет определить реализацию для конкретного набора параметров

```
// general pattern
template <class T>
T abs(T x) { return x > 0 ? x : -x; }

// specialization
template <>
Complex abs(Complex x) {
  return {sqrt(x.re * x.re + x.im * x.im), 0};
}

Complex c{3, 4};
abs(c); // Ok: [T=Complex] {5, 0}
```

```
template <class T> T GetZero() { return 0; }
template <> Complex GetZero() { return {0, 0}; }
```

11.13.9 Параметры шаблона не являющиеся типами

В качестве шаблонных параметров помимо типов могут выступать еще и:

- Целые числа
- Указатели
- Ссылки
- `std::nullptr_t`
- Числа с плавающей точкой (C++20)
- Некоторые классы специального вида (C++20)

```
template <int N, int M>
int Sum() { return N + M; }

Sum<3, 8>(); // Ok: 11

int x = 1;
Sum<1, x>(); // CE (N and M must be compile-time constants!)
```

```
template <class T, size_t N>
size_t ArraySize(const T (&array)[N]) { return N; }

int arr[11];
ArraySize(arr); // Ok: 11 (N is deduced from the type of arr)
```

```
template <void (*FPtr)(int)>
void Call(int x) { FPtr(x); }

Call<f>(10);
```

12 Пользовательские типы

12.1 Перечисления (enum)

12.1.1 Общее представление

Перечисление — пользовательский тип данных, значения которого ограничены набором именованных констант некоторого целого типа.

Проще говоря, перечисление —- тип с некоторым небольшим числом значений.

```
enum Season { kWinter, kSpring, kSummer, kFall };

Season cur_season = kFall; // auto cur_season = kFall;

Season seasons[3] = {kSpring, kFall, kWinter};
Season* ptr = &cur_season; // auto ptr = &cur_season;
Season& ref = cur_season; // auto& ref = cur_season;
```

Значения представляют собой **литералы** (то есть *rvalue*) некоторого целочисленного типа (обычно `int`).

Первое значение равно 0, второе - 1, и т.д.

Значения перечислений неявно преобразуются в низлежащий целый тип, обратные неявные преобразования **запрещены**.

```
static_assert(kWinter == 0); // Ok
static_assert(kFall == 3); // Ok
static_assert(kSpring + kSummer == 3); // Ok
Season s = 0; // CE: implicit conversion prohibited
auto s = static_cast<Season>(0);
```

12.1.2 Правила работы

- Низлежащие значения можно указать явно. Если значение элемента не указано, то его значение равно предыдущему значению + 1.

```
enum Season { kWinter = 10, kSpring, kSummer = -1, kFall };
// kWinter == 10, kSpring == 11, kSummer == -1, kFall == 0
```

- Низлежащий тип тоже можно указать самостоятельно.

```
enum Season : int16_t { kWinter, kSpring, kSummer, kFall };
```

- После определения перечисления можно сразу создать переменные (до `;`)

```
enum Season { kWinter, kSpring, kSummer, kFall} a, *ptr, arr[10];
```

12.1.3 Анонимные перечисления

При объявлении перечисления можно не указывать его имени:

```
enum { kZero, kOne, kThree = 3, kFour, kFive };
```

Такие перечисления использовались для создания именованных констант (константы в языке C появились только в стандарте C89).

Кроме того, в отличие от констант элементы перечислений представляют собой литералы, то есть не требуют отдельного места в памяти.

```
enum { kEnumValue = 11 };
const int kConstValue = 11;

&kConstValue; // valid: kConstValue - variable
&kEnumValue // invalid: kEnumValue - just a value
```

12.1.4 enum class (C++11)

Рекомендованным способом объявления перечислений является `enum class` :

```
enum class Season { kWinter, kSpring, kSummer, kFall };
```

- К значениям `enum class` можно обратиться только по полному имени:

  ```
  auto x = Season::kWinter;
  ```

- `enum class` не допускает неявного приведения к низлежащему типу:

  ```
  int x = Season::kWinter; // CE
  Season::kWinter + Season::kFall; // CE
  ```

- Как и к пространствам имен к ним применима директива `using` :

  ```
  using enum Season; auto x = kWinter; // Ok
  ```

12.2 Объединения (union)

12.2.1 Общие представления

Объединение — пользовательский тип данных, который в каждый момент времени позволяет хранить объект одного из заранее указанных типов в общей области памяти.

Проще говоря, это тип, который может хранить разнородные данные (но только один объект в каждый момент).

```
union IntDouble {
  int i;
  double d;
};
```

12.

- Так как объединения хранят данные в одной и той же области памяти, то размер объединений равен наибольшему из размеров своих членов:

  ```
  sizeof(IntDouble) == max(sizeof(int), sizeof(double));
  ```

- Обратиться к членам объединения можно с помощью операции `.` :

```
IntDouble x;
x.i = 0; // current member - i
x.d = 1.5; // and now d
```

- Так как объединения хранят данные в одной и той же области памяти, то адреса его членов равны (C++14)

```
IntDouble x;
static_cast<void*>(&x.i) == static_cast<void*>(&x.d)
```

- Действующий член - член объединения, к которому в последний раз была применена операция(operator) записи. Чтение из не действующего члена - *UB*!

```
IntDouble x;
x.i = 0; // current member - i
std::cout << x.d; // UB
```

- Существуют анонимные объединения. Они позволяют объявить переменные с разделяемой областью памяти:

```
union { int x; bool y; }; // variables x and y are in the same memory
x = 1;
y = false;
```

- Объединения могут иметь статические члены. Работают они так же, как и статические члены структур (про них в следующем разделе)

12.3 Структуры (struct)

12.3.1 Общее представление

Структура — составной тип данных, который инкапсулирует набор данных (возможно разных типов) в одном объекте.

Пример 1: Комплексное число - это тип, состоящий из двух значений:

```
struct Complex {
  double re; // real part
  double im; //imaginary part
};
```

Пример 2: Тип, который вместе с указателем хранит размер массива, на который он указывает:

```
struct Array {
  int* buffer;
  size_t size;
};
```

12.3.2 Правила работы

Рассмотрим следующую структуру и на ней покажем правила работы:

```
struct S {
  int x;
  float y;
  const char* z;
};
```

- Как и массивы, структуры можно инициализировать фигурными скобками (если значения для каких-то полей не указаны, то они обнуляются):

```
S a; // struct is not initialized
S b = {1, 2, "string"};
S c{1}; // x == 1, y == 0, z == nullptr
```

- Обращение к полям происходит с помощью операции `.` :

```
S a;
a.x = 11;
std::cout << a.x;
```

- После определения структуры (перед `;`) можно сразу описать переменные:

```
struct S {
  int x;
  float y;
} a = {1, 2.0}, *ptr, &ref = a, arr[10];
```

- При копировании структуры побитово копируют все свои поля. Кроме того, таким образом можно даже скопировать один массив в другой!

```
struct S {
  float x;
  int arr[3];
};

S a = {1.0, {1, 2, 3}};
S b = a; // all fields of a are copied bit by bit to b
a = b; // assignment works too
```

- Чтобы обратиться к полю структуры через указатель можно сделать так:

```
// create a structure in dynamic memory
auto ptr = new S;

// dereference the pointer, refer to the field x
(*ptr).x = 0;
```

Последнюю строчку можно заменить более простым синтаксисом:

```
ptr->x = 0; // operator ->
```

- При создании массива структур применима агрегатная инициализация. Как всегда, непроинициализированные части заполнятся нулями:

```
struct S {
  int x;
  float y;
};

S array[3]{{1, 2}, 3}; // {{1, 2}, {3, 0}, {0, 0}}
```

- Полям можно задать значения по умолчанию:

```
struct S {
  int x = 1;
  float y;
};

S a; // a.x == 1, a.y == ???
```

- Поля могут быть константными. Значения таких полей должны быть указаны во время инициализации

```
struct S {
  int x;
  const float y; };

  S a; // CE
  S b = {2}; // Ok (b.y == 0)
  S c = {1, 2}; // Ok (c.y == 2)
```

- Допускается не более одного определения одной и той же структуры на единицу трансляции. При этом все определения в программе должны быть **идентичны**.

12.3.3 Статические поля

Некоторые поля структур могут быть помечены словом `static` :

```
struct S {
  static int x;
};
```

Такие поля принадлежат не конкретному объекту, а **всей структуре**. То есть это поле является общим для всех объектов и хранится в статической (глобальной) области.

```
S a;
S b;

a.x = 1;
b.x == 1; // true
b.x = 2;
a.x == 2; // true
```

К статическим полям можно обращаться напрямую через имя класса (оно ведь принадлежит классу в целом, а не конкретному объекту):

```
S::x = 0;
std::cout << S::x;
```

Но есть нюанс. Статические поля являются видом глобальных переменных, а для переменных ODR требует ровно одного определения во всей программе. Поэтому определять статические переменные внутри структур нельзя. Исключение - константные целочисленные поля:

```
struct S {
  static const int x = 1; // it is possible
  static int y = 2; //    it is impossible
};
```

Для того, чтобы такое было возможно, необходимо использовать `inline` :

```
struct S {
  inline static int x = 1;
};
```

12.3.4 Размер стуктуры. Выравнивание

Размер структуры не вычисляется как сумма занимаего места полями.

Так, например:

```
struct S {
  char c; // 1
  int32_t i; // 4
  int16_t s; // 2
  int64_t l; // 8
};

std::cout << sizeof(S) << '\n'; // 24

struct    {
  int64_t l; // 8
  int16_t s; // 2
  char c; // 1
  int32_t i; // 4
};

std::cout << sizeof(C) << '\n'; // 16
```

Правила работы с памятью звучат следующим образом:

- Размер структуры не меньше суммы размеров ее полей и не меньше 1.
- Поля структуры идут в памяти строго в порядке перечисления в определении.
- Каждый объект в C/C++ выравнивается по адресу кратному его размеру. То есть значение `&x` кратно `sizeof(x)` .
- Наиболее эффективным (по занимаемой памяти) расположением полей будет упорядочивание по убыванию размера.

  Однако на практике не стоит заниматься такими микрооптимизациями.
- При необходимости структуры дополняются байтами в конце, чтобы при создании массива структур все его элементы были выровнены.

Можно принудительно выровнять все поля по 1 байту:

```
#pragma pack(push, 1)
struct S {
  char c; // 1
  int32_t i; // 4
  int16_t s; // 2
  int64_t l; // 8
};
#pragma pack(pop)

std::cout << sizeof(S) << '\n'; // 15
```

Но так делать не стоит!

13 Парадигмы программирования

- **Парадигма программирования** — это совокупность идей и понятий, определяющих стиль написания компьютерных программ.
- Служит для упрощения разработки и поддержки кода программ.
- При проектировании и написании кода парадигма отвечает на вопрос "как?".
- Язык программирования может поддерживать сразу несколько парадигм.

13.1 Примеры

- **Императивное программирования** — программирование с описанием последовательности инструкций, ветвлений, безусловных переходов, иногда с вызовом подпрограмм.

  Языки: *Asm, C/C++, Fortran, Basic, Pascal*, ·
- **Структурное программирование** — программирование с использованием независимых логически законченных процедур/функций.

  Языки: *C/C++, Fortran, Basic, Pascal, Java* ·
- **Объектно-ориентированное программирование** — парадигма основанная на представлении программы в виде совокупности классов, объектов и их взаимодействий.

  Языки: *C++, Java, Python,* ·

13.2 Принципы ООП

- **Абстракция** — выделение наиболее важных свойств объектов реального мира и их оформление в виде атрибутов класса.
- **Инкапсуляция** - объединение данных и методов для работы с ними в рамках одного объекта, возможно с ограничением доступа к деталям реализации (сокрытие данных).
- **Полиморфизм** - свойство системы, позволяющее использовать различные реализации в рамках одного интерфейса (один интерфейс - много реализаций).
- **Наследование** - свойство, позволяющее создавать новый тип данных на основе уже существующего с полным или частичным заимствованием функционала.

14 Классы и объекты

14.1 Класс

Класс — описание некоторого концепта из предметной области в виде набора полей и методов для работы с ними (описание нового типа данных).

```
struct Stack {
  inline static const size_t kMaxSize = 100;
  size_t size = 0;
  int buffer[kMaxSize];

  // method declarations:
  void Push(int value);
  void Pop();
  int Top();
  void Clear();
};
```

14.2 Объект

Объект — экземпляр класса.

```
Stack stack; // create object
std::cout << stack.size << '\n'; // 0
stack.Push(1);
std::cout << stack.Top() << '\n' // 1
```

14.3 Модификаторы доступа

14.3.1 Синтаксис

Синтаксис выглядит достаточно просто. Перед блоком полей/методов необходимо указать модификатор:

```
struct Stack {
  // ...
private:
  // ...
public:
  // ...
};
```

14.3.2 Правила использовани

- `public` — любой внешний по отношению к классу код имеет доступ к полям и методам.
- `private` — доступ имеют только поля и методы самого класса, а также дружественные функции и классы (обсудим позже).
- Располагать модификаторы доступа внутри класса можно в любом порядке и в любом количестве.
- Модификатор действует с точки объявления до следующего модификатора (либо до конца класса).

- В структурах по умолчанию работает модификатор `public` .

14.3.3 Пример

```
struct S {
  int x;
  void g();
  void h(int);
private:
  int y;
  void f();
  void h(double);
};

int main() {
  S s;
  s.x = 0; // Ok
  s.y = 1; // CE
  s.f(); // CE
  s.g(); // Ok
  s.h(0); // Ok
  s.h(0.0); // CE
  return 0;
}
```

14.4 Ключевое слово class

- Для создания нового типа (класса) вместо `struct` можно использовать ключевое слово `class` .

```
class Stack {
  // ...
};
```

- Классы **полностью** эквивалентны структурам, но есть два нюанса:
  1. В классах модификатор доступа по умолчанию — `private` , в структурах — `public` .
  2. Классы наследуют по умолчанию приватным образом, структуры — публичным.
- Как правило, предпочитают использовать `class` . Структуры обычно пишут без методов и они состоят только из открытых полей базовых типов (POD-типы).

14.5 Определение вне классов

- Методы можно определять вне классов (обычно в отдельном cpp), при этом они все еще должны быть предварительно объявлены внутри тела класса

```
class S {
  int x = 0;
  void f();
};

void S::f() { std::cout << x; }
```

- При определении методов внутри класса они автоматически становятся `inline`

```
class S {
  int x = 0;
  void f() { std::cout << x; } // inline
};
```

14.6 Константные методы

Рассмотрим класс стека:

```
class Stack {
private:
  inline static const size_t kMaxSize = 100;
  size_t size_;
  int buffer_[kMaxSize];

public:
  void Push(int value) { buffer_[size_++] = value; }
  void Pop() { --size_; }
  int Top() { return buffer_[size_ - 1]; }
  void Clear() { size_ = 0; }

  size_t Size() { return size_; }
  bool Empty() { return size_ == 0; }
};
```

Метод `Size()` не изменяет состояние нашего стека, но при использовании его в константном объекте мы получим *CE*:

```
const Stack cs;
cs.Size(); // CE

error: passing 'const␣Stack' as 'this' argument discards qualifiers
```

Поэтому компилятору нужно объяснить, что данный метод ничего не меняет:

```
size_t Stack::Size() const { // <-- !
  return size;
}

cs.Size(); // Ok
```

Это не только подсказка для компилятора, но и указание, что в методе **нельзя изменять поля**!

```
size_t Stack::Size() const {
  return ++size; // CE
}

error: increment of member 'Stack::size' in read-only object
```

14.6.1 Правила работы

- В константных методах можно вызывать только константные методы (неконстантные могут изменить поля)

```
size_t Stack::Size() const {
  // CE: calling a non-const method in a const
  Pop();
  return size;
}

int Stack::Top() const {
  // Ok: calling a constant method
  return buffer[Size() - 1];
}
```

- Константность — часть сигнатуры, поэтому по константности можно делать перегрузку

```
// called for const stacks
int Stack::Top() const {
  return buffer[size - 1]; }

// called for non-const stacks
int& Stack::Top() {
  return buffer[size - 1];
}

Stack a;
const auto& cref = a;
// ...
// non-constant stack now allows you to change the top
a.Top() = 1;
cref.Top() = 1; // CE

std::cout << cref.Top() << a.Top() << '\n'; // OK
```

14.7 Статические методы

14.7.1 Общий подход

Некоторые методы можно сделать статическими:

```
class S {
public:
  int x;
  static int y;

  void f();
  static void g();
};
```

14.7.2 Правила работы

- Такие методы принадлежат не конкретному объекту, а классу в целом.
- К ним можно обращаться не только через объект класса, но и через имя класса с помощью операции `::` (`S::y` или `S::g()`).
- Статические методы могут работать только со статическими полями (в противном случае непонятно, какому объекту принадлежит поле)

14.7.3 Пример

Рассмотрим следующий код:

```
class S {
public:
  int x;
  static int y;

  void f() {
    x += 1;
    y += 1;
  }

  static void g() {
    y += 1;
  }
};

int main() {
  S a, b;
  a.x = 0; b.x = 1; // a.x = 0, b.x = 1
  a.y = 0; b.y = 1; // a.y = 1, b.y = 1
  S::x = 10; // CE
  S::y = 11; // a.y = 11, b.y = 11
  a.f(); b.f(); // a.x = 1, b.x = 2
  a.g(); b.g(); // a.y = 15, b.y = 15
  S::f(); // CE
  S::g(); // a.y = 16, b.y = 16
  return 0;
}
```

Важно!
Константных статических методов не существует.

14.8 Итоговый вид класса стек

Используя все знания выше получим корректный код стека на массиве:

```
class Stack {
private:
  inline static const size_t kMaxSize = 100;
  size_t size_;
  int buffer_[kMaxSize];

public:
  void Push(int value) { buffer_[size_++] = value; }
  void Pop() { --size_; }
  int& Top() { return buffer_[size_ - 1]; }
  void Clear() { size_ = 0; }

  int Top() const { return buffer_[size_ - 1]; }
  size_t Size() const { return size_; }
  bool Empty() const { return size_ == 0; }

  static size_t MaxSize() { return kMaxSize; }
};
```

14.9 Логическая и физическая константность

14.9.1 Виды константности

Существует 2 вида константности объектов:

- Объект **логически константен**, если с точки зрения пользователя объект не меняет своего состояния (нельзя отличить объект до и после операции).
- Объект **физически константен**, если его внутреннее представление никак не меняется (не изменилось ни одного бита внутри объекта).

Пример: для отладки объект логирует информацию о действиях над собой (считает число вызовов метода). С физической точки зрения объект не константа (обновляется счетчик), с логической - константа (пользователь не наблюдает этих изменений).

14.9.2 Ключевое слово mutable

Рассмотрим для примера класс:

```
class C {
  size_t counter = 0;

 public:
  int GetZero() {
    ++counter;
    return 0;
    }
};

const C c;
c.GetZero(); // CE
```

метод `GetZero()` не может быть константным в таком виде, так как он нарушает физическую константность объекта. Но можно сказать компилятору, что логическая константен.

Чтобы сообщить компилятору, что изменение данного поля не влияет на логическую константность объекта, его можно пометить ключевым словом `mutable` .

`mutable` поля можно изменять в константных методах.

```
class C {
  mutable size_t counter = 0;

 public:
  int GetZero() const {
    ++counter;
    return 0;
    }
};

const C c;
c.GetZero(); // OK
```

15 Шаблоны классов

15.1 Синтаксис и применение

Синтаксиси шаблонов класса аналогичен шаблонам функций.

Но для данных классов действует ряд следующих правил:

- В отличие от обычных классов можно объявлять только в области видимости пространства имен, либо внутри другого класса.
- Тип шаблонного параметра нужно указывать явно (C++17: если невозможно вывести тип по конструктору).
- Шаблоны классов (как и другие шаблоны) инстанцируются "лениво". Более того, методы шаблонного класса тоже инстанцируются "лениво".

```
template <class T>
class S {
 public:
  void f(); // instantiated
  void g(); // dont instantiated
};

S<int> s;
s.f();
```

15.2 Специализация шаблонов

Как и в случае шаблонов функций шаблон и специализация могут кардинально отличаться (разные поля, разные методы):

```
// general pattern
template <class T>
struct IsInt {
   static const bool value = false;
};

// complete specialization
template <> struct IsInt<int> {
  static const bool value = true;
};

template <>
struct IsInt<bool* const> {
  void Print() const { std::cout << "No"; }
};
```

15.3 Частичная специализация шаблонов классов

Иногда хочется задать определенное поведение класса не для конкретного типа, а для целого семейства типов (например, для указателей). Такая срециалазация называется *частичной*.

```cpp
template <class T> // general pattern
struct IsPointer {
  static const bool value = false;
  static bool IsIntPointer() { return false; }
};

template <class T> // partial specialization
struct IsPointer<T*> {
  static const bool value = true;
  static bool IsIntPointer() { return false; }
};

template <> // complete specialization
struct IsPointer<int*> {
  static const bool value = true;
  static bool IsIntPointer() { return true; }
};
```

Более специализированная версия всегда побеждает менее специализированную.

15.4 Шаблонные параметры шаблонов

Шаблон может принимать другие шаблоны в качестве параметров.

```cpp
template <class T>
class Array {
  // ...
};

// the template that takes a type and a template that takes a type
template <class T, template <class> class Container>
class Stack {
  Container<T> buffer;

 public:
 // ...
};

Stack<int, Array> stack;
```

16 Конструкторы

Конструктор — особый метод класса, который вызывается при создании объекта.

Свойства конструктора:

- Этот метод не имеет возвращаемого значения.
- Его имя *совпадает* с именем класса.

- Конструктор вызывается *неявно* при создании объекта.

```
class Stack {
 public:
  Stack() {
    buffer_ = new int[kCapacity];
    size_ = 0;
  }
  // ... (other methods)
};

Stack stack; // The Stack::Stack() constructor is called
```

Существует несколько видов конструкторов:

- Параметрический конструктор
- Конструктор преобразования
- Конструктор по умолчанию
- Конструктор копирования
- Конструктор перемещения (будет рассмотрен в курсе позднее)

16.1 Параметрический конструктор

Параметрический конструктор — конструктор, который может быть вызван с несколькими аргументами (больше 1).

```
class Stack {
 public:
  Stack(size_t size, int value) { // stack of size elements of value
    buffer_ = new int[kCapacity];
    size_ = size;
    for (size_t i = 0; i < size_; ++i) {
      buffer_[i] = value;
    }
  }
  // ... (other methods)
};

Stack stack(10, 1);
// or Stack stack = Stack(10, 1);
// or auto stack = Stack(10, 1);
stack.Size(); // 10
stack.Top(); // 1
```

16.2 Списки инициализации

Дело в том, что в момент выполнения тела конструктора все поля уже должны быть проинициализированы. Таким образом, нельзя инициализировать поля внутри тела конструктора. Значит нужно инициализировать поля еще до входа в тело конструктора. В особенности если у нас есть поля, которые являются ссылками или константами.

Требуемая синтаксическая конструкция называется *списком инициализации*.

```
class B {
  const int x_;
  double& y_;

 public:
  B(int x, double& y) : x_(x), y_(y) { // initialize
    // Working with initialized data
  }
};

double z = 0.0;
B b(0, z); // Ok
```

Если список инициализации пуст, то все поля инициализируются по умолчанию.

Если какое-то поле не проинициализировано, компилятор попытается проинициализировать его самостоятельно.

Важное правило: порядок создания полей определяется порядком их объявления в классе, а не порядком в списке инициализации

16.3 Конструктор преобразования

Конструктор преобразования - конструктор, который может принимать ровно 1 аргумент.

- Данный конструктор используется для неявных (или явных) преобразований.
- Стандарт C++11 расширил понятие "конструктор преобразования но это уже совсем другая история.

```
Stack::Stack(size_t size) : buffer_(new int[kCapacity]{}), size_(size)
}

Stack stack(1); // stack size 1
Stack another = 3; // stack size 3

void f(Stack arg);

f(stack); // Ok
f(10); // Ok <=> f(Stack(10))
```

16.3.1 Ключевое слово explicit

Иногда требуется запретить неявное преобразование нашего класса, чтобы избежать поведения, которое не предполагается.

Чтобы запретить подобные неявные (implicit) преобразования, необходимо попросить, чтобы они выполнялись только явно (`explicit`).

```
explicit Stack::Stack(size_t size) { ... }

Stack stack(1); // Ok: explicit conversion
Stack another = 3; // CE: implicit conversion

void f(Stack arg);
f(stack); // Ok: no conversion
f(10); // CE: implicit conversion
```

16.4 Конструктор по умолчанию

Конструктор по умолчанию — конструктор, который может быть вызван без аргументов.

```
Stack::Stack() : buffer_(new int[kCapacity]), size_(0) {
}
Stack stack;
// or auto stack = Stack();

Stack other(); // CE: the compiler treats this as a function declaratio
```

Если вы не объявляете *явно* ни одного конструктора, то компилятор создаст для класса свой конструктор по умолчанию, который инициализирует все поля-классы конструкторами по умолчанию, а для полей-примитивных типов ничего не делает.

Но, если в классе вы явно объявили хотя бы один конструктор, то никакого неявного конструктора по умолчанию компилятор создавать не будет.

Конструктор по умолчанию предоставляемый компилятором эквивалентен конструктору с пустым телом:

```
class A { ... };

class B {
 public:
  int x;
  A a;
  B() {} // <=> B() : a() {}
};
```

Напоминание: список инициализации выполняется **в любом случае**.

Компилятор также может отказаться генерировать свой конструктор по умолчанию, если он не понимает как это сделать.

```
struct A {
 private:
  A() {}
};

struct B {
  A a; // Oops: private constructor A()
  const int b; // Oops: how to initialize a constant?
  int& c; // Oops: how to initialize a reference?
};

B b; // Compilation error
```

16.4.1 Конструкция =default

Допустим, в вашем классе уже есть какой-то конструктор, а писать конструктор по умолчанию самостоятельно не хочется.

В этом случае можно явно попросить компилятор предоставить свою версию конструктора по умолчанию с помощью `= default` .

Такой конструктор эквивалентен конструктору с пустым телом.

```
struct B {
  B(int x, int y) {}
  B() = default; // equal B() {}
};
```

16.5 Конструктор копирования

Конструктор копирования — конструктор, который создает объект с помощью другого объекта того же типа путем его копирования.

Как следует из определения, единственный аргумент - объект того же типа.

Однако тривиальная передача по значению обречена на провал:

```
Stack::Stack(Stack s) : buffer_(new int[kCapacity]), size_(s.size_) {
  for (size_t i = 0; i < size_; ++i) {
    buffer_[i] = s.buffer_[i];
  }
}

Stack a;
Stack b(a); // Endless recursion!
```

Для того, чтобы избежать этого следует передавать аргумент по константной ссылке. Так будет корректно работать конструктор с анонимными объектами и мы никак не испортим исходный объект.

```
Stack::Stack(const Stack& other) { ... }

Stack a;
Stack b(a); // Ok

const Stack a;
// ...
Stack b(a); // Ok
Stack   (Stack()); // Ok
```

Если вы не объявляете своего конструктора копирования (и конструктора перемещения), то компилятор создаст для класса свой конструктор копирования, который поля-классы инициализируются конструкторами копирования, а поля-базовые типы копируются побитово:

```
class A { ... };
class B {
  A a;
  int x;
  // the gift from the compiler
  // B(const B& other) : a(other.a), x(other.x) {}
};
```

Но компилятор откажется это делать, если в классе есть поля, которые нельзя скопировать.

```
struct A {
  A() = default;

 private: A(const A&);
};

struct B {
  A a;
};

B first;
B second(first); // Compilation error
```

16.5.1 Конструкция =default

Аналогично конструктору по умолчанию можно *явно* попросить компилятор создать свою версию конструктора копирования с помощью `= default` .

```
struct B {
  B() = default;
  B(const B&) = default;
};

B first;
B second(first); // Ok
```

Хотя это не является обязательным — достаточно просто не написать свой конструктор копирования, чтобы компилятор сгенерировал свой.

Замечание

Для создания дефолтного конструктора копирования недостаточно определить его с пустыми фигурными скобками:

```
class A { ... };

struct B {
  int x;
  A a;

  B() = default; // <=> B() : a() {}
  B() {}; // <=> B() : a() {}

  B(const B&) = default; // <=> B(const B&) : x(other.x), a(other.a) {}
  B(const B&) {} // <=> B(const B&) : a() {}};
```

16.6 Делегирующие конструкторы

Часто так бывает, что конструкторы дублируют действия друг друга. Поэтому хотелось бы иметь возможность вызывать конструктор в другом конструкторе. И в этом помогут *делегирующие конструкторы*:

```
Stack::Stack(size_t size, const int* values) :
  buffer_(new int[kCapacity]) ,
  size_(size) {

  for (size_t i = 0; i < size; ++i) {
    buffer_[i] = values[i];
  }
}

Stack::Stack(const Stack& other) : Stack(other.size_, other.buffer_) {
}
```

Если применено делегирование, то инициализация полностью на совести вызванного конструктора, отдельные поля проинициализировать не получится.

17 Деструктор

Деструктор - особый метод класса, который вызывается при завершении времени жизни объекта.

Основная информация:

- Этот метод не имеет возвращаемого значения и аргументов.
- Его имя = `~<ClasesName>` .
- Деструктор вызывается неявно при уничтожении любого объекта, однако может быть вызван и явно (как метод). Но так делать не стоит (как правило, приводит к UB).

```
Stack::~Stack() {
  delete[] buffer_;
}

Stack stack;
stack.Push(1);
// Ok, no memory leaks
```

17.1 Правила работы с деструктором

- Если вы не пишите своего деструктора, то компилятор создаст для класса свой, который вызывает деструкторы для полей-классов, а для полей-базовых типов ничего не делает.
- Если невозможно вызвать деструктор у какого-либо поля, то компилятор откажется создавать свой деструктор.
- Чтобы явно указать намерение использовать предоставленный компилятором деструктор, можно использовать `= default` .
- Если деструктор недоступен, то вы не сможете создавать объекты на стеке и в статической области памяти.
- Но можно создавать объекты в динамической области. Правда, в этом случае придется управлять памятью на низком уровне.

17.2 Что делать в деструкторе

- Если при уничтожении объекта не требуется выполнения нетривиальных действий, то ничего (лучше предоставить работу компилятору).
- Если уничтожение объекта требует освобождения выделенной памяти, закрытия файлов, логирования и т.д., прописывайте эти действия в деструкторе (компилятор не догадается самостоятельно вызвать `delete` !)
- Важно помнить, что, хотите вы того или нет, у каждого поля при выходе из тела деструктора вызовется свой деструктор, поэтому вручную уничтожать поля не нужно.

```
Stack::~Stack() {
  delete[] buffer_; // Ok
  size_ = 0; // Ok, but why?
};

struct B {
  Stack s;
  ~B() {
    s.~Stack(); // UB: do not do it this way
  } // <-- Stack destructor will be called here again
};
```

18 Конструкторы и деструкторы

18.1 Идиома RAII

Конструкторы и деструкторы позволяют реализовать важнейшую идиому языка C++ — **RAII** (*Resource Acquisition Is Initialization/Захват Ресурса - это Инициализация*)

Идея в том, чтобы выделение и освобождение ресурса происходило автоматически (в конструкторе и деструкторе соответственно).

```
auto ptr = new int(10); // not safe! You can forget delete

// RAII
class IntPtr {
  int* ptr;
 public:
  explicit IntPtr(int value) : ptr(new int(value)) {}
  ~IntPtr() { delete ptr; }
  // ...
};
```

18.2 Порядок вызова конструкторов/деструкторов

Стековые объекты создаются в порядке объявления, а уничтожаются в обратном

```
struct A {
  A() { std::cout << "A()␣"; }
  ~A() { std::cout << "~A()␣"; }
};

struct B {
  B() { std::cout << "B()␣"; }
  ~B() { std::cout << "~B()␣"; }
};

int main() {
  A a;
  B b;
  return 0;
}
```

В итоге получим следующие вывод — `A() B() ~B() ~A()`

18.3 Правило трех

В C++ существует правило приличия, которое носит название правило трех:

"Если класс требует реализации пользовательского деструктора либо конструктора копирования, либо операции присваивания, то требуется реализовать все эти три сущности"

Несмотря на то, что это правило не является правилом языка (ошибки компиляции нарушение этого правила не провоцирует), важно следовать этому правилу для корректной работы ваших классов

Далее в курсе это правило эволюционирует в правило пяти и правило ноля.

18.4 Конструкция = delete

Начиная с C++11, можно объявлять функции "удаленными". Такие функции нельзя вызывать, а также нельзя получать указатель на них.

```
void f(int);
void f(double) = delete;

template <class T>
void g(T);
template <>
void g(int) = delete;
// ...

f(1); // Ok
f(1.0); // CE
g(1); // CE
g(1.0); // Ok
```

- Как правило, эта возможность используется для запрета генерации некоторых методов (например, для запрета копирования).

```
struct C {
  // remove the copy constructor
  C(const C&) = delete;
  // ...
};
```

- Альтернативно, можно объявить метод приватным. (плохой стиль)

```
struct C {
 private:
  C(const C&) = delete;
  // ...
};
```

19 Прегрузка операция(operator)

В C++ есть механизм, позволяющий определить поведение операций при работе с классами, структурами и перечислениями. Этот механизм называется *перегрузкой операций*.

Перегрузка операций наравне с перегрузкой функций и шаблонами функций является примером проявления *статического полиморфизма*.

19.1 Синтаксис

Синтаксис перегрузки операций совпадает с определением функций: сначала идет тип возвращаемого значения, затем имя операции (`operator<op>` , `<op>` — символ операции), далее список аргументов операции и тело операции.

```
Complex operator+(const Complex& x, const Complex& y) {
  return Complex(x.re + y.re, x.im + y.im);
}

Complex(1, 2) + Complex(3, 4); // Ok: 4 + 6i
// Equivalently operator+(Complex(1, 2), Complex(3, 4))
```

Альтернативно, можно объявить операцию как член класса, а не как внешнюю функцию. В этом случае текущий объект автоматически является левым операндом, и в качестве аргумента достаточно передать только правый (при необходимости)

```
Complex Complex::operator+(const Complex& y) const {
  return Complex(re + y.re, im + y.im);
}

Complex(1, 2) + Complex(3, 4); // Ok: 4 + 6i
// Equivalently operator+(Complex(1, 2), Complex(3, 4))
```

19.2 Перегрузка унарных операций

Некоторые операции принимают ровно 1 аргумент (унарный `+`, унарный `-`, унарная `*` и т.д.).

В этом случае соответствующая перегрузка тоже принимает 1 аргумент.

```
// Implementation as an external function
Complex operator-(const Complex& x) {
  return {-x.re, -x.im};
}
// Implementation as a class method
Complex Complex::operator-() const {
  return {-re, -im};
}
```

19.3 Правила и особенности

Существует несколько правил, которые необходимо учитывать при реализации перегрузки операция(operator):

1. Нельзя переопределять операции с примитивными типами (хотя бы один из аргументов должен быть пользовательского типа).

   ```
   const char* operator*(const char* str, int n); // CE
   Complex operator+(const Complex& x, int y); // Ok
   ```

2. Нельзя вводить новые операции в язык (например, `**` для возведения в степень)

   ```
   Complex operator**(const Complex& x, int n); // CE
   ```

3. Нельзя менять арность и приоритет операций.

4. Нельзя переопределять операции `::`, `.`, `?:`, `.*`.

5. Операции `=`, `()`, `[]`, `->` могут быть перегружены только в виде методов.

6. Операции `&&` и `||` теряют свойство "короткого вычисления (до C++17 теряли строгий порядок вычисления вместе с операцией `,`).

При выборе реализации операции для класса нужно помнить, что при реализации перегрузки методом класса, левый операнд всегда определен, а значит невозможно использовать неявные пребразования для него. Если такое необходимо, то стоит использовать реализацию как внешнюю функцию.

19.4 Перегрузка пре- инкремента и декремента

Операции префиксного инкремента и декремента - унарные операторы, которые (для базовых типов) изменяют значение переменной на +1/-1 и *возвращают ссылку на ту же переменную.*

Это стоит учитывать при перегрузке операций для своего класса (хотя, конечно, компилятор этого проверять не будет)

```
// 1 option
Complex& operator++(Complex& value) {
  ++value.re;
  return value;
}

// 2 option
Complex& Complex::operator++() {
  ++re;
  return *this;
}
```

19.5 Перегрузка пост- инкремента и декремента

Существуют постфиксные версии инкремента и декремента. Их отличие от префиксных в том, что они возвращают копию старого значения переменной.

Чтобы перегрузить постфиксную операцию надо (внезапно!) добавить фиктивный аргумент типа `int` .

```
// 1 option
Complex operator++(Complex& value, int) {
  auto old_value = value;
  ++value.re;
  return old_value;
}

// 2 option
Complex Complex::operator++(int) {
  auto old_value = *this;
  ++re;
  return old_value;
}
```

19.6 Перегрузка операции индексирования

операция(operator) индексирования (aka операция(operator) доступа к элементу массива) может быть перегружена только *в форме метода класса* и может иметь *только один аргумент*.

```
// only for reading
int IntArray::operator[](size_t i) const {
  return buffer_[i];
}

// read and write
int& IntArray::operator[](size_t i) {
  return buffer_[i];
}

// And that's not possible
int& Matrix::operator[](size_t i, size_t j);
```

19.7 Перегрузка круглых скобок

операция(operator) "круглые скобки"(aka операция(operator) функционального вызова) может быть перегружена только *в форме метода класса* и может иметь *сколько угодно аргументов*.

```
struct Printer {
  void operator()(int i, const char* str, char c) const {
    std::cout << i << '␣' << str << '␣' << c; }
};

Printer print;
print(1, "Hello", '+'); // 1 Hello +
```

19.8 Перегрузка операции присваивания

- Классическая операция(operator) присваивания присваивает значение правого аргумента левому аргументу и возвращает ссылку на левый операнд.
- Ваша реализация не обязана удовлетворять этому соглашению, но мир станет лучше, если вы будете следовать этим правилам.
- операция(operator) присваивания одна из четырех операций, которая может быть реализована только как метод класса.
- операция(operator) присваивания - часть "правила трех".

```
Complex& Complex::operator=(const Complex& other) {
  re = other.re;
  im = other.im;
  return *this;
}
```

Если вы не реализуете свой оператор присваивания компилятор создаст за вас свой, который присваивает каждое поле поотдельности и возвращает ссылку.

То есть код выше излишний - без него оператор присваивания был бы создан автоматически и делал бы то же самое.

Как и ранее, можно явно попросить компилятор создать свою версию:

```
Complex& Complex::operator=(const Complex& other) = default;
```

19.8.1 Проблема самоприсваивания

Рассмотрим следующую реализацию присваивания стека

```
Stack& Stack::operator=(const Stack& other) {
  delete[] buffer_;
  buffer_ = new T[kCapacity];
  size_ = other.size_;
  for (size_t i = 0; i < size_; ++i) {
    buffer_[i] = other.buffer_[i];
  }
  return *this;
}

Stack a;
// ...
a = a;
```

В данном случае у нас возникнет проблема с тем, что мы сначала удалим массив из динамической памяти, а после попробуем им воспользоваться. Для того чтобы избежать этой проблемы, нужно добавить проверку в начале:

```
if (this != &other) { ... }
```

19.8.2 Составные операции присваивания

Составные операции присваивания имеют вид `+=` , `-=` , `*=` и т.д. В языке C++ они изменяют левый операнд и возвращают ссылку на левый операнд.

В отличие от простого присваивания на перегрузку этих операторов не накладывается ограничений (это обычный бинарный оператор).

Однако, как правило, его реализуют как член класса и возвращают ссылку на левый операнд

```
// 1 option
Complex& Complex::operator+=(const Complex& other) {
  re += other.re;
  im += other.im;
  return *this;
}

// 2 option (as opposed to operator=)
Complex& operator+=(Complex& lhs, const Complex& rhs) {
  lhs.re += rhs.re;
  lhs.im += rhs.im;
  return lhs;
}
```

19.9 Перегрузка побитового сдвига

Так сложилось, что побитовый сдвиг переопределяют не по прямому назначению, а для целей ввода из потока и вывода в поток.

```
std::cin >> x;
std::cout << x;
```

Объект `std::cin` является объектом класса `std::istream`, который осуществляет считывание потока из консоли (cin = Console INput).

Объект `std::cout` является объектом класса `std::ostream`, который осуществляет запись потока в консоль (cout = Console OUTput).

Заголовочный файл `<iostream>` включает множество других классов потоков, в том числе `std::ifstream` и `std::ofstream` (ввод/вывод в файл), `std::istringstream` и `std::ostringstream` (ввод/вывод в строку).

```
std::istringstream sin("1␣2␣3");
sin >> x >> y >> z; // x = 1, y = 2, z = 3

std::ostringstream sout;
sout << x << ",␣" << y << ",␣" << z;
sout.str(); // returns "1, 2, 3"
```

Где-то в `<iostream>` перегружена операция(operator) `>>`, где в качестве левого операнда принимается объект типа `std::istream`, а в качестве правого — `int`.

Кроме того, так как мы хотим, чтобы работал следующий код:

```
int x;
double y;
std::cin >> x >> y; //Same as (std::cin >> x) >> y;
```

Необходимо, чтобы `>>` возвращал левый операнд (поток). Тогда после ввода `x` вернется `std::cin` и вызовется следующая операция(operator) с `y`.

```
std::istream& operator>>(std::istream& is, int& value);
std::ostream& operator<<(std::ostream& os, int value);
```

19.9.1 Ключевое слово friend

Так как очень часто для ввода и вывода операции побитового сдвига необзходим будет доступ к приватным полям, то необходимо объявить такие функции "друзьми"класса.

Друзья класса - внешние по отношению к классу сущности, которые имеют доступ к приватной и защищенной частям класса.

- Можно объявлять другие функции другом класса:

```
class Stack {
  // ...
  friend void Print(const Stack& stack);
};

void Print(const Stack& stack) {
  for (size_t i = 0; i < stack.size; ++i) {
    cout << stack.buffer[i] << '␣';
  }
}
```

- Друга можно сразу объявить и определить (создать):

```
class Stack {
  // ...
  friend void Print(const Stack& stack) { ... }
};
```

- Другом может быть метод другого класса, а также другой класс целиком:

```
class A {
  void f() { B b; b.g(); b.j(); } // Ok
  void h() { B b; b.g(); b.j(); } // CE: h() not a friend for B
  friend class B;
};

class B {
  void g() { A a; a.f(); a.h(); } // Ok
  void j() { A a; a.f(); a.h(); } // Ok
  friend A::f();
};
```

19.9.2 Реализация перегрузки

Исходя из всего вышесказанного перегрузка побитового сдвига чаще всего будет выглядеть:

```
class Complex {
  double re_;
  double im_;

  // ...
  friend std::istream& operator>>(std::istream& is, Complex& complex);
};

std::istream& operator>>(std::istream& is, Complex& complex) {
  is >> complex.re >> complex.im;
  return is;
}
```

20 Итераторы

На данный момент интераторы стоит воспринимать как обертку над указателями, для которых также определена операция(operator) разыменования. И у каждого контейнера (речь о них пойдет дальше), есть:

- Итератор на начало — `c.begin()` , где `c` — контейнер.
- Итератор на следующий после последнего элемента — `c.end()` , где `c` — контейнер.

С итераторами можно выполянть те же операции, что и с указателями (см. арифметика указателей).

21 Последовательные контейнеры

Контейнер - тип данных, который обеспечивает хранение объектов других типов, а также интерфейс для доступа к ним.

Последовательный контейнер обеспечивает упорядоченный способ хранения элементов, завиясящий от времени и места их добавления, но не зависящий от их значений.

21.0.1 std::array

`std::array<T, N>` - аналог C-style массива из `N` элементов типа `T` .

Все элементы живут на стеке, что обеспечивает быстрый доступ к ним, но лишает возможности динамического расширения.

21.0.2 Правила работы

При использовании `std::array` важно знать, что:

- Может быть быть проинициализирован так же как и обычный массив:

```
int c_style[5]{1, 2, 3}; // [1, 2, 3, 0, 0]
std::array<int, 5> cpp_style{1, 2, 3}; // [1, 2, 3, 0, 0]
```

- При отсутствии инициализатора заполняется значениями по умолчанию:

```
std::array<int, 5> ints; // no initialization
ints[0]; // UB

std::array<SomeClass, 5> objects; // 5 default constructors
objects[0]; // Ok
```

- В отличие от C-style массивов могут быть скопированы.

```
int copy[5] = c_style; // CE
auto copy = cpp_style; // Ok
```

- Могут быть переданы в функцию по значению.

```
void GetCArray(int arr[5]); // <=> void GetCArray(int* arr);
void GetArray(std::array<int, 5> arr);

GetCArray(c_style); // a pointer to the first element is passed!
GetArray(cpp_style); // is copied into the entire argument
```

- Чтобы избежать копирования можно принимать по ссылке

```
void GetByRef(const std::array<int, 5>& arr);

GetByRef(cpp_style);
```

21.0.3 Операции и методы

Доступные операции и методы, которые можно использовать:

- Для доступа к элементам используются `[]`, `at`, `front`, `back`, `data`. Время работы каждого метода или операции — $\mathcal{O}(1)$
  - `[]` — небезопасный доступ по индексу (без проверки, что выходит за границы).
  - `at` — безопасный доступ по индексу (при выходе за границу выбрасывается исключение).
  - `front` — возвращает ссылку на первый элемент.
  - `back` — возвращает ссылку на последний элемент.
  - `data` — возвращает указатель на первый элемент.
- Методы, связанные с размером контейнера — `empty`, `size`. Время работы каждого метода — $\mathcal{O}(1)$
  - `empty` — возвращает `true`, в случае если контейнер пуст или `false` в ином случае.
  - `size` — возвращает количество элементов

```
void IWantPointer(int* ptr, int size);
IWantPointer(array.data(), array.size());
```

- Методы `fill` и `swap`. Время работы каждого метода — $\mathcal{O}(n)$

```
array.fill(1); // [1, 1, 1, 1, 1]

array.swap(copy);
// array == [1, 2, 3, 0, 0]; copy == [1, 1, 1, 1, 1]
```

- Также есть логическое сравнение. Сравниваются лексикографически. Время работы — $\mathcal{O}(n)$

21.1 std::vector

`std::vector<T>` — динамически расширяющийся массив элементов типа `T`.

В отличие от `std::array` имеет нефиксированный размер и хранит данные не на стеке, а в динамической памяти (куче).

21.1.1 Инициализация

Инициализировать `std::vector<T>` можно различными способами:

```
// empty array
std::vector<T> a;

// array of n elements created by default
std::vector<T> b(n);

// array of n copies of value
std::vector<T> c(n, value);

// list initialization of an array
std::vector<T> d{1, 2, 3};

// copy [O(N)]
auto b_copy = b;

// move [O(1)]
auto c_move = std::move(c);
```

21.1.2 Доступ к элементам

Доступные методы для работы с элементами:

- Для доступа к элементам используются `[]`, `at`, `front`, `back`, `data`. Время работы каждого метода или операции — $\mathcal{O}(1)$

- `[]` — небезопасный доступ по индексу (без проверки, что выходит за границы).
- `at` — безопасный доступ по индексу (при выходе за границу выбрасывается исключение).
- `front` — возвращает ссылку на первый элемент.
- `back` — возвращает ссылку на последний элемент.
- `data` — возвращает указатель на первый элемент.

- Методы, связанные с размером контейнера — `empty` , `size` . Время работы каждого метода — $\mathcal{O}(1)$
  - `empty` — возвращает `true` , в случае если контейнер пуст или `false` в ином случае.
  - `size` — возвращает количество элементов

```
void IWantPointer(int* ptr, int size);
IWantPointer(vactor.data(), vector.size());
```

- Методы `fill` и `swap` . Время работы `fill` — $\mathcal{O}(n)$, а `swap` — $\mathcal{O}(1)$.

```
vector.fill(1); // [1, 1, 1, 1, 1]

vector.swap(copy);
```

- Также есть логическое сравнение. Сравниваются лексикографически. Время работы — $\mathcal{O}(n)$

21.1.3 Вставка и удаление элементов

- Вставка и удаление с конца — `push_back` и `pop_back` . Амортизационно работают за $\mathcal{O}(1)$

```
std::vector<int> v{1, 2};
v.push_back(3); // {1, 2, 3}
v.pop_back(); // {1, 2}
```

- Вставка и удаленье в любое место — `insert` , `erase` . В худшем случае работают за $\mathcal{O}(n)$

```
std::vector<int> v{1, 2, 3};

v.insert(v.begin() + 1, 0); // {1, 0, 2, 3}
v.insert(v.begin() + 3, 2, -1); // {1, 0, 2, -1, -1, 3}

v.erase(v.begin() + 1); // {1, 2, -1, -1, 3}
v.erase(v.begin() + 2, v.begin() + 4); // {1, 2, 3}
```

- Изменение размера вектора — `resize(count)` . Работает за $\Omega(|size - count|)$

```
std::vector<int> v{1, 2};
v.resize(4); // {1, 2, ?, ?}
v.resize(5, -1); // {1, 2, ?, ?, -1}
v.resize(2); // {1, 2}
```

- Очистка ветора (но не удаление) — `clear` . Работает за $\Theta(size)$

```
std::vector<int> v{1, 2};
v.clear(); // {}
```

21.1.4 Управление хранилищем

Для обеспечения амортизационной константы во времени добавления элементов в конец используется мультипликативная схема расширения массива.

В частности, это означает, что в каждый момент времени памяти выделено несколько больше, чем нужно для хранения всех объектов.

В связи с этим у вектора есть 2 разных атрибута: `size` и `capacity` (`capacity` $\geqslant$ `size`).

Методы для управления хранилища:

- `reserve(count)` — увеличивает вместимость массива минимум до count. Работает за $\mathcal{O}(size)$

```
std::vector<int> v(5); // v.size() == 5, v.capacity() >= 5
v.reserve(10); // v.size() == 5, v.capacity() >= 10
v.reserve(2); // Nothing happens
```

- `shrink_to_fit` — уменьшает `capacity` до `size` .

```
std::vector<int> v(100); // v.size() == 100, v.capacity() >= 100
v.resize(1); // v.size() == 1 , v.capacity() >= 100
v.shrink_to_fit(); // v.size() == 1 , v.capacity() == 1
```

`shrink_to_fit` — единственный метод, который может уменьшить `capacity` (`resize` и `reserve` до меньших размеров не приводят к реаллокации).

21.1.5 Полезные советы

1. Используйте `reserve` если вам известно количество элементов.

```cpp
//bad style
const auto n = ReadInt();
std::vector<int> v;
for (int i = 0; i < n; ++i) {
  v.push_back(ReadInt()); // periodic memory reallocations
}

//good style
const auto n = ReadInt();
std::vector<int> v;
v.reserve(n); // immediately allocate memory for n elements
for (int i = 0; i < n; ++i) {
  v.push_back(ReadInt()); // reallocation does not occur
}
```

2. Используйте erase-remove идиому. Например, нам нужно удалить все 0 в массиве.

```cpp
//bad style
for (int i = 0; i < v.size();) { // O(n^2)
  if (v[i] == 0) {
    v.erase(v.begin() + i);
  } else {
    ++i;
  }
}

//good style
auto zeros_begin = std::remove(v.begin(), v.end(), 0); // O(n)
v.erase(zeros_begin, v.end()); // O(n)
```

21.1.6 Инвалидация ссылок и указателей

В силу того, что хранилище массива может менять свой размер, ему необходимо перевыделять память. А значит место в памяти изменяется при изменении самого массива, значит следующий код не считается валидным:

```cpp
std::vector<int> v;
// ...
int& x = v[5];
v.push_back(10); // reallocation occurred
x = 0; // BOOM! (reference no longer valid)

std::vector<int> v;
//...
int* p = &v[5];
v.push_back(10); // reallocation occurred
*p = 0; // BOOM! (pointer sag)
```

21.2 std::deque

`std::deque<T>` - шаблонный класс двунаправленной очереди.

Позволяет эффективно (амортизированно за $\mathcal{O}(1)$) добавлять элементы как в начало, так и в конец (`push_back` , `push_front` , `emplace_back` , `emplace_front`).

В остальном похож на вектор (`[]` , `at` , `resize` , `clear` , ...).

В силу внутреннего устройства не предоставляет методы `reserve` , `capacity` , нопозволяет вызывать `shrink_to_fit` .

21.2.1 Инвалидация ссылок и указателей

В отличии от вектора, в деке нет проблемы перевыделения памяти, при котором будет происходить инвалидация ссылок или указателей.

```
std::deque<int> d;
// ...
int& x = d[5];
d.push_back(10); // reallocation occurred
x = 0; // Always OK

std::deque<int> d;
//...
int* p = &d[5];
d.push_back(10); // reallocation occurred
*p = 0; // Always OK
```

21.3 std::deque vs std::vector

В общем случае, операции над `std::vector` эффективнее чем над `std::deque` .

Если речь идет о возможности добавления в начало и инвалидации ссылок, то стоит предпочесть `std::deque` .

И помните главный принцип C++: "не плати за то, что не используешь".

21.4 std::list

`std::list<T>` - шаблонный класс двусвязного списка.

21.4.1 Методы

- Работа с элементами в конце — `push_back` , `emplace_back` , `pop_back` .
- Работа с элементами в начале — `push_front` , `emplace_front` , `pop_front` .
- Доступ к первому и последнему элементам — `front` , `back` . Доступ к остальным элементам осуществляется только через итераторы. У листа нет операции доступа по индексу (`[]`)
- Методы вставки и удаления — `insert` , `erase` . Работают за $\mathcal{O}(1)$.

Лист также не инвалидирует ссыли и указатели на элементы.

21.4.2 slice

Преимуществом списков является то, что можно быстро (за $\mathcal{O}(1)$) перенести элементы одного списка в другой или переставить элементы списка внутри него самого.

```
std::list<int> l{...};
std::list<int> other{...};
```

- Splice

```
// move all nodes from other to l, to position pos
l.splice(pos, other);
```

- Single-element splice

```
// move the node pointed to by elemmove the node pointed to by elem
l.splice(pos, other, elem);
```

- Range splice

```
// move nodes from begin to end
l.splice(pos, other, begin, end);
```

21.5 std::forward_list

`std::forward_list<T>` — шаблонный класс односвязного списка.

Ясно, что с односвязным списком можно делать те же операции, что и над двусвязным, за исключением `push_back` , `pop_back` .

Имея итератор на элемент двусвязного списка, можно свободно его удалить, или вставить перед ним элемент

Для односвязного списка указателя на узел не достаточно для его удаления или вставки перед ним.

Можно лишь вставить или удалить элемент после него. Поэтому для всех методов в конце добавляется суффикс `_after` . Кроме того, у `std::forward_list<T>` нет `size` .

21.6 Типы-члены

Каждый контейнер определяет внутри себя ряд типов, которые можно использовать для извлечения его свойств.

- `value_type` — тип хранимых элементов
- `size_type` ==- в чем измеряется размер (что возвращает `size`)
- `difference_type` — в чем измеряется расстояние между объектами
- `reference` — тип ссылки на элемент
- `const_reference` — тип ссылки на константный элемент

- `pointer` — тип указателя на элемент
- `const_pointer` — тип указателя на константный элемент

Допустим, нужна функция, которая принимает контейнер и копирует содержимое контейнера в динамический массив. Тогда, зная о типах-членах мы можем легко написать такую функцию:

```
template <class Container>
typename Container::pointer ToCstyleArray(const Container& container) {
  auto array = new typename Container::value_type[container.size()];
  // copy...
  return array;
}
```

22 Функторы

Функторы (или функциональные объекты) - объекты, для которых определена операция(operator) функционального вызова (функции, указатели на функцию, объекты с перегруженным `operator()`).

22.1 Примеры

Примеры функторов могут быть различными, вот некоторые из них:

- Передача пользовательского делитера в умный указатель:

```
std::unique_ptr<FILE, int(*)(FILE*)> file_ptr(std::fopen(
    "filename", "wr"), std::fclose);
```

- Передача компаратора в `std::sort` :

```
struct SizeCompare {
  bool operator()(const std::string& lhs,
    const std::string& rhs) const {
    return lhs.size() < rhs.size();
  }
};

std::sort(v.begin(), v.end(), SizeCompare{});
```

22.2 Стандартная библиотека

В стандартной библиотеке определено небольшое количество функторов, которые могут быть использованы как параметры в других алгоритмах. (ссылка)

Наиболее часто используемые из них: `std::less` (<), `std::greater` (>), `std::equal_to` (==), `std::plus` (+), `std::minus` (-).

22.3 Лямбда выражения (C++11)

Проблема компараторов через классы — слишком много лишнего кода. Для решения этой проблемы к нам на помощь приходят лямбда выражения.

22.3.1 Синтаксис

- В начале пишется список захвата - `[...]` .
- Затем в `(...)` аргументы функции (произвольное количество). Если аргументов нет, то можно опустить.
- В конце в `{...}` идет тело функции.

```
auto f = [](int x) { std::cout << x << '␣'; };
auto g = [](int x, double y) {
  std::cout << x << '␣' << y << '␣'; return x; };

auto h = [] { std::cout << "empty";};

f(g(1, 2.0)); //1 2 1
h(); // empty
```

22.3.2 Возвращаемые тип

Заметьте, что до этого мы не прописывали тип возвращаемого значения.

Дело в том, что в случае одного (или нуля) `return` выражения этот тип можно легко вывести (по правилам вывода шаблонного параметра).

```
[] {}; // void
[](const std::string& lhs, const std::string& rhs) { // bool
  return lhs < rhs;
};
```

Если `return` выражений несколько, и они не противоречат друг другу, то тип тоже выводится без проблем:

```
[](int x) { if (x > 0) return; }; // void

[](const std::string& lhs, const std::string& rhs) { // bool
  if (lhs < rhs) return true;
  return false;
};
```

Если `return` выражений несколько, и типы возвращаемых значений различны, то возникает ошибка компиляции:

```
// void or int?
[](int x) { if (x > 0) return; return 0; }; // CE

// bool or int?
[](const std::string& lhs, const std::string& rhs) { // CE
  if (lhs < rhs) return true;
  return 0;
};
```

В случае конфликта, его можно указать явно после списка аргументов:

```
[](const std::string& lhs, const std::string& rhs) -> bool {
  if (lhs < rhs) return true;
  return 0;
};
```

Кстати, этот же синтаксис можно использовать и с обычными функциями:

```
auto main() -> int {
  return 0;
}
```

22.3.3 Аргументы

В случае аргументов, единственное, что достойно упоминания: шаблонные лямбда выражения.

Если хочется, чтобы выражение работало с произвольными типами (или лень писать полный тип аргументов), можно на месте типа арумента писать auto (с любыми квалификаторами).

```
std::sort(v.begin(), v.end(), [](const auto& lhs, const auto& rhs) {
  return lhs.size() < rhs.size();
});
```

Лямбда выражение при этом аналогично объявлению:

```
// T and U do not have to match
template <class T, class U>
auto lambda(const T&, const U&);
```

22.3.4 Список захвата

Внутри лямбда выражений можно свободно использовать глобальные переменные, переменные из глобальных пространств имен, а также статические переменные функции.

Однако локальные переменные внутри лямбда выражения не видны.

Для использования локальных переменных функций их нужно захватить.

Для захвата по значению нужно просто написать имя переменной в списке захвата `[...]`. Если хочется захватить несколько переменных, их нужно перечислить через запятую.

После этого внутри лямбда выражения будет сожержаться неизменяемая копия захваченного объекта.

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4};
  [x, v]() { std::cout << x << '␣' << v[2]; }(); // immediately call
}
```

Для захвата всех локальных переменных по значению, можно написать `[=]` (Плохой стиль). А чтобы появилась возможность изменения захваченных копий, можно написать `mutable` перед телом лямбды:

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4}; // <-- this vector will not change!
  [=]() mutable { v[0] = x; std::cout << v[0] << v[1] << v[2]; }();
}
```

Для захвата локальных переменных по ссылке необходимо перед именем переменной написать `&` . Переменные, на которые ссылается лямбда можно изменять.

Для захвата всех локальных переменных по ссылке, можно написать `[&]` . (Плохой стиль)

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4}; // <-- this vector will change!
  [&]() { v[0] = x; std::cout << v[0] << v[1] << v[2]; }();
}
```

Можно захватить все переменные по значению и некоторые по ссылке, и наоборот:

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4}; // <-- this vector will change!
  [=, &v]() { v[0] = x; }(); // capture everything by value except v
}
```

Можно задать имя, по которому можно обращаться к захваченной переменной:

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4};
  [value = x, &buf = v, &const_buf = std::as_const(v)]() {
    buf[0] = value;
    const_buf[0] = value; // CE, const_buf - constant reference to v
  }();
}
```

Если же мы хотим использовать поля класса в лямбда выражении, то мы можем захватить `this` (Плохой стиль) или же использовать захват с переименованием:

```cpp
struct S {
  int x;
  void f() {
    [x = this->x]() { std::cout << x; }();
  }
};
```

23 Ассоциативные контейнеры

Ассоциативный контейнер - контейнер, обеспечивающий быстрый поиск элементов.

В C++ ассоциативные контейнеры представлены шаблонными классами:

- `std::set` / `std::multiset`
- `std::map` / `std::multimap`
- `std::unordered_set` / `std::unordered_multiset`
- `std::unordered_map` / `std::unordered_multimap`

23.1 std::set / std::multiset

`std::set<KeyT, Compare = std::less<KeyT>>` — множество уникальных ключей.

`std::multiset<KeyT, Compare = std::less<KeyT>>` — множество ключей.

- `Compare` - функтор, т.е. тип, у которого определена операция(operator) `()` , принимающая два объекта `KeyT` и возвращающая `true` , если первый операнд меньше (строго!) второго.
- Элементы упорядочены по ключу согласно сравнению `Compare` .
- Обеспечивают логарифмический поиск, вставку и удаление элементов.
- Как правило, реализованы с помощью бинарного дерева поиска.

23.1.1 Конструкторы

Есть несколько видов конструкторов множества:

```cpp
std::set<int> s; // empty set

std::array<int, 5> arr{1, 2, 3, 1, 3};
std::set<int> s(arr.begin(), arr.end()); // {1, 2, 3}

std::array<int, 5> arr{1, 2, 3, 1, 3};
std::multiset<int> s(arr.begin(), arr.end()); // {1, 1, 2, 3, 3}

std::set<int, std::greater<>> s{5, 2, 7, 1, 5}; // {7, 5, 2, 1}
```

23.1.2 Поиск

Поиск элементов осуществляется с помощью основных методов:

- `count(key)` — число элементов `key` во множестве. Работает за $\mathcal{O}(logn + \#key)$.
- `find(key)` — итератор на элемент `key` (`end()` , если ключа нет). Работает за $\mathcal{O}(logn)$

  Важно! Менять значение ключа по итератору нельзя

  `*s.find(5) = 10; //CE`
- `lower_bound(key)` — итератор на первый элемент >= `key` . Работает за $\mathcal{O}(logn)$
- `upper_bound(key)` — итератор на первый элемент > `key` . Работает за $\mathcal{O}(logn)$
- `equal_range(key)` — пара (`std::pair`) `lower_bound(key)` , `upper_bound(key)` . Работает за $\mathcal{O}(logn)$

23.1.3 Вставка

Вставка элементов происходит следующим образом:

- `insert(key)` — вставка элемента `key` , возвращает пару успех/неуспех и итератор на элемент. Работает за $\mathcal{O}(logn)$.

```
auto [success, it] = s.insert(5);
success; // inserted/not inserted (true/false)
it; // iterator
```

-
- `emplace(Args&&...)` — вставляет элемент с параметрами конструктора `Args&&...` , возвращает пару успех/неуспех и итератор на элемент. Работает за $\mathcal{O}(logn)$
- `insert(begin, end)` — вставляет последовательность элементов. Работает за $\mathcal{O}(klog(n+k))$

  `s.insert(arr.begin(), arr.end());`
- При вставке можно передать "подсказку— итератор на ближайший элемент *больший* вставляемого.

  То есть, если дано множество $\{1, 2, 4\}$, то при вставке 3 можно дополнительно передать итератор на элемент 4. При вставке 0 подсказка — итератор на *begin*. При вставке 5 подсказка — итератор на *end*. В случае верной подсказки вставка будет работать амортизировано за $\mathcal{O}(1)$. Иначе $\mathcal{O}(logn)$.

  – `insert(hint, key)` — вставка `key` , `hint` — итератор, подсказка, кудавставить. Возвращает итератор на элемент. Работает за $\mathcal{O}(1)$ - $\mathcal{O}(logn)$.

23.1.4 Удаление

Для удаления необходимо использовать следующие вариации `erase` :

- `erase(iterator)` — удаляет элемент по итератору, возвращает итератор на следующий по величине элемент. Работает за $\mathcal{O}(logn)$.
- `erase(begin, end)` — удаляет последовательность элементов. Работает за $\mathcal{O}((logn + distance(begin, end)))$
- `erase(key)` — удаляет все элементы с ключом `key` , возвращает число удаленных элементов. Работает за $\mathcal{O}(logn + \#key)$.

23.1.5 Прочее

- `empty()` , `size()` , `swap()` . Работают за $\mathcal{O}(1)$.
- `clear()` . Работает за $\mathcal{O}(n)$.
- При обходе от `begin()` до `end()` элементы перечисляются в порядке возрастания.

23.1.6 Изменение ключа

Изменить ключ в ассоциативном контейнере нереально, можно только лишь удалить и вставить новый.

```
s.erase(old_key); // remove the old key and node
s.insert(new_key); // create a new node and key
```

Начиная с *C++17* можно извлечь узел из контейнера, изменить в нем в ключ и вставить обратно.

```
auto node = s.extract(old_key); // extract node
node.value() = new_key; // change key in node
s.insert(std::move(node)); // insert back
```

23.2 std::map / std::multimap

`std::map<KeyT, ValueT, Compare = std::less<KeyT>>` — отображение из множества уникальных ключей в значения.

`std::multimap<KeyT, ValueT, Compare = std::less<KeyT>>` — множество пар "ключ-значение".

- `Compare` - функтор, т.е. тип, у которого определена операция(operator) `()` , принимающая два объекта `KeyT` и возвращающая `true` , если первый операнд меньше (строго!) второго.
- Пары "ключ-значение"упорядочены по ключу согласно сравнению `Compare` .
- Обеспечивают логарифмический поиск, вставку и удаление элементов.
- Как правило, реализованы с помощью бинарного дерева поиска.

23.2.1 Методы

Методы аналогичны `std::set` и `std::multiset` , только в качестве параметров передаются пары "ключ-значение":

```
std::array<std::pair<int, std::string>, 3> arr[3]{
  {1, "one"},
  {2, "two"},
  {3, "three"}
};

std::map<int, std::string> m(arr.begin(), arr.end());
m.insert(std::make_pair(4, std::string("four")));

auto it = m.find(3);
it->first; // key == 3
it->second; // value == "three"
```

23.2.2 Обращение к элементам

`operator[](key)` (только для `std::map`) — возвращает ссылку на значение, связвнное с ключом `key` .

Важно!

Если ключа нет, то он вставляется и создается значение по умолчанию (если `ValueT` — базовый тип, то инициализируется нулем!).

Поэтому нельзя использовать для константных мап, а также для мап, в которых `ValueT` не имеет конструктора по умолчанию.

```
std::map<int, int> m;
const std::map<int, int> cm;
//...
m[0]; // Ok
cm[0]; // CE
```

Кроме того, плохой стиль написания:

```
std::map<std::string, int> counter;
std::string str = GetString();
if (counter.find(str) != counter.end()) {
  ++counter[str];
} else {
  counter.emplace(str, 1);
}
```

Хороший:

```
std::map<std::string, int> counter;
std::string str = GetString();
++counter[str];
```

`at(key)` (кроме `std::multimap`) — возвращает ссылку на значение, связанное с ключом `key` .

Важно! Если ключа нет, то бросается исключение `std::out_of_range` .

Может быть использован в константных мапах:

```
const std::map<int, int> cm{{1, 1}, {2, 2}, {3, 3}};
std::cout << cm[1]; // CE
std::cout << cm.at(1); // Ok
```

23.2.3 Изменение ключа

Работает аналогично `std::set`

```
auto node = m.extract(old_key); // extract node
node.key() = new_key; // change key in node
m.insert(std::move(node)); // insert back
```

23.3 std::unordered

Контейнеры выглядят следующим образом:

`std::unordered_set<KeyT, Hash = std::hash<KeyT>, Equal = std::equal_to<KeyT>>`

`std::unordered_map<KeyT, ValueT, Hash = std::hash<KeyT>, Equal = std::equal_to<KeyT>>`

Аналогичны `std::set` , `std::map` , `std::multiset` , `std::multimap` , но в среднем более эффективный поиск, вставка и удаление [O(1)], так как основаны на хеш таблицах.

Однако не хранят порядок элементов. Поэтому отсутствуют методы `lower_bound` , `upper_bound` , `equal_range` . Обход осуществляется в произвольном порядке.

23.3.1 Дополнительные методы

- `float load_factor() const` — возвращает отношение числа элементов к числу корзин.
- `float max_load_factor() const` — возвращает `load_factor` при котором происходит перехеширование (увеличение числа корзин).
- `void max_load_factor(float)` — устанавливает `load_factor` , при котором должно происходить увеличение числа корзин.
- `rehash(count)` — изменить число корзин на `count` .
- `reserve(count)` — устанавливает число корзин достаточное для хранения `count` элементов.
- `bucket(key)` — номер корзины для элемента `key` .
- `bucket_size(id)` — размер корзины `id` .
- `bucket_count()` — число корзин

24 Контейнеры адаптеры

Контейнерный адаптер — обертка над контейнером, предоставляющая ограниченный интерфейс для работы с данными.

В стандартный библиотеке контейнерные адаптеры представлены тремя шаблонными классами:

- `std::stack`
- `std::queue`
- `std::priority_queue`

24.1 std::stack

`std::stack` `std::stack<T, Container = std::deque<T>>` — предоставляет интерфейс стека.

Хранит данные типа `T` в контейнере `Container` и использует методы `Container` для реализации своих.

- *Конструкторы:* по умолчанию, copy, конструктор от контейнера
- *Методы:* `top()`, `empty()`, `size()`, `push(const T&)`, `push(T&&)`, `emplace(Args&&...)`, `pop()`, `swap()`.

`Container` должен содержать методы `back`, `push_back`, `emplace_back`, `pop_back`, `empty`, `size`, а также типы-члены `value_type`, `size_type`, `reference`, `const_reference`.

```
std::stack<int> s;
s.push(1);
s.push(2);
s.top(); // 2
s.pop();
s.top(); // 1

std::stack<int, std::vector<int>> s(std::vector<int>{1, 2, 3});
s.top(); // 3
```

Возможная реализация:

```cpp
template <class T, class Container = std::deque<T>>
class stack {
  Container container_;
 public:
  using const_reference = typename Container::const_reference;
  // ... other member types

  stack() = default;
  stack(const Container& container): container_(container) { }

  void push(const T& value) {
    container_.push_back(value);
  }
  const_reference top() const {
    return container_.back();
  }
  // ... other methods };
```

24.2 std::queue

`std::queue<T, Container = std::deque<T>>` — предоставляет интерфейс очереди.

Хранит данные типа `T` в контейнере `Container` и использует методы `Container` для реализации своих.

- *Конструкторы:* по умолчанию, copy, конструктор от контейнера
- *Методы:* `back()`, `empty()`, `size()`, `push(const T&)`, `push(T&&)`, `emplace(Args&&...)`, `pop()`, `swap()`.

`Container` должен содержать методы `back`, `front` `push_back`, `emplace_back`, `pop_back`, `empty`, `size`, а также типы-члены `value_type`, `size_type`, `reference`, `const_reference`.

24.3 std::priority_queue

`std::priority_queue<T, Container = std::vector<T>, Compare = std::less<T>>` — предоставляет интерфейс очереди.

Хранит данные типа `T` в контейнере `Container` и использует методы `Container` для реализации своих. На вершине очереди находится наибольший элемент с точки зрения `Compare`.

- *Конструкторы:* по умолчанию, copy, конструктор от контейнера, конструктор от компаратора, конструктор от компаратора и контейнера.
- *Методы:* `top()`, `empty()`, `size()`, `push(const T&)`, `push(T&&)`, `emplace(Args&&...)`, `pop()`, `swap()`.

`Container` должен содержать методы `front` `push_back`, `emplace_back`, `pop_back`, `empty`, `size`, `[]` а также типы-члены `value_type`, `size_type`, `reference`, `const_reference`.

25 Structured bindings (структурные связывания)

Объявление структурного связывания - это всегда auto объявление (возможно с cv или ссылочными квалификаторами), в котором идентификаторы перечислены в [...] через запятую:

```
auto [x, y] = f();
auto&& [a, b, c] = g();
const auto& [z] = h();
```

25.1 Инициализация

Структурные связывания могут быть проинициализированы:

- С-массивами

```
int array[3]{1, 2, 3};
auto [x, y, z] = array;
```

- Объектами классов с открытыми нестатическими полями (например, `std::pair`)

```
auto [x, y] = std::make_pair(1, 2);
```

- Объектами классов, для которых определены `std::tuple_size`, `std::tuple_element`, `std::get` (например, `std::tuple`)

```
auto [x, y, z] = std::make_tuple(1, std::string("aba"), 2.0);
```

26 initializer list

`std::initializer_list` - шаблонный класс, представляющий легковесную обертку над константным массивом объектов.

Список объектов в фигурных скобках автоматически преобразуются в `std::initializer_list` в следующих ситуациях:

- При создании объекта с помощью объявления auto:

```
auto l = {1, 2, 3}; // type(l) == std::initializer_list<int>
```

- При создании объекта класса, поддерживающего конструктор от `std::initializer_list`:

```
Stack<T>::Stack(std::initializer_list<T>);
Stack<int> s{1, 2, 3}; // Ok
```

- При передаче в функцию, принимающую в качестве аргумента `std::initializer_list` :

```
void f(std::initializer_list<int>);
f({1, 2, 3});
```

`std::initializer_list` содержит всего 3 метода: `size()` , `begin()` , `end()` .
Категория итератора - random access (contiguous, начиная с C++20)
Типы-члены: `value_type` , `reference` , `const_reference` , `size_type` , `iterator` , `const_iterator`
Пример использования:

```
template <class T>
Stack<T>::Stack(std::initializer_list<T> lst)
    : buffer_(new T[lst.size()])
    , size_(0)
    , capacity_(lst.size()) {
  for (auto&& obj : lst) {
    Push(obj);
  }
}
```

27 Наследование и полиморфизм

Наследование - свойство, позволяющее создавать новый тип данных на основе уже существующих с полным или частичным заимствованием функционала.

Наследование является одним из основных понятий, на которых зиждется ООП.

27.1 Пример и синтаксис наследования

Допустим пишем простую компьютерную игру с некоторым набором персонажей, которые могут сражаться друг с другом и, возможно, выполнять другие действия:

```
struct Archer {
  int strength;
  int hp;
  int xp;

  void Shoot();
  // ...
};

struct Soldier {
  int strength;
  int hp;
  int xp;
  void Melee();
  // ...
};
```

Ясно, что с увеличением числа персонажей и их возможностей код может разрастаться очень сильно.

```
class Hero;

class Archer;
class Melee;
class Cavalry;
class Imposter;
```

Таким образом, возникает две ключевые проблемы:

1. Дублирование кода.
2. Отсутствие ясной структуры и иерархии классов.

Для решения проблем дублирования кода и указания связи между классами можно воспользоваться следующим синтаксисом:

```
struct Hero {
  int strength;
  int hp;
  int xp;

  void Heal();
  // ...
};

// archer is inherited from class "character"
struct Archer : public Hero {
  void Shoot();
  // ... (other specific methods and fields)
};
```

`Hero` — базовый класс, `Archer` — производный класс (класс-наследник).
Теперь все поля и методы `Hero` также являются полями и методами класса `Archer` :

```
Hero hero;
Archer archer;

hero.hp = 90; // hp - hero field
archer.hp = 80; // archer also has such a field

hero.Heal(); // likewise
archer.Heal();

archer.Shoot(); // Archer is a more specific class
hero.Shoot(); // CE
```

27.2 Модификаторы доступа и наследование

27.2.1 public и private

`public` и `private` модификаторы работают как и раньше — `public` открывает доступ всем, `private` закрывает доступ для всех, даже для наследников

```
struct Hero {
 public:
  void Heal() { UpdateHp(20); }
 private:
 void UpdateHp(int delta) { hp += delta; }
};

struct Archer : public Hero {
  void Rest() {
    Heal(); // Ok (public)
    UpdateHp(10); // CE (private)
  }
};

Archer archer;
archer.Heal(); // Ok (public)
archer.UpdateHp(10); // CE (private)
```

`public` и `private` могут задавать и режим наследования, то есть задавать уровень доступа к полям и методам базового класса.

- *public наследование* — все знают о том, что класс унаследован от базового, и внешний код имеет полный доступ к открытым полям и методам базового класса.
- *private наследование* — никто не должен знать о наследовании, доступ к открытым полям и методам базового класса имеет только наследник.

```
struct A {
 public: int x;
 private: void f();
};

struct B : public A {
  void h() {
    x = 0; // Ok
    f(); // CE
  }
};

struct C : private A {
  void h() {
    x = 0; // Ok
    f(); // CE
  }
};

A a;
a.x = 11; // Ok
a.f(); // CE

B b;
b.x = 11; // Ok
b.f(); // CE

C c;
c.x = 11; // CE
c.f(); // CE
```

27.2.2 protected

Существует третий модификатор доступа — `protected` .

Он работает так же как и `private` , но доступ дополнительно получают наследники класса:

```
struct A {
 protected:
  int x;
  void f();
};

struct B : public A {
  void h() { x = 0; f(); } // Ok
};

A a;
a.x = 0; a.f(); // CE
B b;
b.h();
b.x = 0; b.f(); // CE
```

`protected` можно использовать для изменения режима наследования.
protected наследование — никто не имеет доступ к базовому классу, кроме наследников

```
struct A {
 public: int x;
 private: void f();
 protected: void g();
};

struct B : protected A {
  void h() {
    x = 0; // Ok
    f(); // CE
    g(); // Ok
  }
};

struct C : public B {};

A a;
a.g(); // CE

B b;
b.x = 11; // CE
b.f(); // CE
b.g(); // CE
b.h(); // Ok

C c;
c.x = 11; // CE
c.f(); // CE
c.g(); // CE
c.h(); // Ok
```

27.2.3 Семантики при наследовании

- `public` режим наследования позволяет обращаться к полям и методам базового класса напрямую (то есть пользоваться наследником в точности как базовым классом). Реализует семантику "является".
- `private` и `protected` режимы запрещают внешнему коду (кроме друзей) каким-либо образом использовать знание о том, что что-то от чего-то унаследовано. `protected` дополнительно разрешает доступ для наследников класса (для остальных работает как `private`). Реализует семантику "содержит".

27.2.4 Empty Base Optimization (EBO)

`private` и `protected` наследование почти всегда можно заменить на композицию (введение поля нужного типа в класс):

```
struct A { /* ... */ };

struct B : private A { /* ... */ };

struct C {
 private: A a;
 // ...
};
```

Классы C и B практически эквивалентны (с точки зрения внешнего кода эквивалентны, так как он не использует факт наследования от A или наличия поля a).

Как мы знаем, размер в байтах любого, даже пустого, объекта в C++ обязан быть > 0.

Однако при наследовании от пустого класса размер наследника не увеличивается

```
struct A {}; // sizeof(A) == 1

struct B : private A { // sizeof(B) == sizeof(int)
  int x;
};

struct C { // sizeof(C) > sizeof(int)
  int x;
 private:
  A a;
};
```

Это бывает полезно, если класс реализован с помощью другого класса, у которого нет нестатических полей. В этом случае наследование лучше композиции.

27.3 Отличие class от struct

Как было известно ранее — в классах модификатор доступа по умолчанию - private , в структурах - public.

С наследованием аналогично: классы по умолчанию наследуют приватно, а структуры - публично.

```
class A { /* ... */ };

struct S : A { /* ... */ };
// <=>
struct S : public A { /* ... */ };

class C : A { /* ... */ };
// <=>
class C : private A { /* ... */ };
```

На этом список отличий заканчивается.

27.4 Порядок вызова конструкторов при наследовании

Класс инициализируется в следующем порядке: сначала инициализируются базовые классы, затем поля класса-наследника в порядке объявления (список инициализации не может повлиять на порядок!)

```
struct A {
  int x;
  int y;
  A(int);
  A(int, int);
};

struct B : public A {
  int z = 0;
  B() {} // CE, A has no default constructor
  B(int x) : A(x) {}
  B(int x, int y) : A(x, y) {} // first A(x, y), then z = 0
  B(int x, int y, int z) : A(x, y), z(z) {}
};
```

Чтобы не дублировать конструкторы базового класса (как в прошлом примере) можно воспользоваться конструкцией `using A::A` (теперь B можно создавать так же как и A)

```
struct A {
  int x;
  int y;
  A(int);
  A(int, int);
};

struct B : public A {
  int z = 0;

  using A::A;
  B(int x, int y, int z) : A(x, y), z(z) {}
};
```

Важно!

Нельзя проинициализировать отдельное поле базового класса, только всю базовую часть целиком.

27.5 Порядок вызова деструкторов

При уничтожении объекта сначала выполняется тело деструктора, затем деструктурируются поля класса-наследника в порядке обратном объявлению и в самом конце уничтожаются части базового класса

```
class A {
  // ...
};

class B : public A {
  Stack s1;
  Stack s2;

  // ...
  ~B() {
    // (1) ...
  } // <- (2) s2.~Stack(), (3) s1.~Stack(), (4) ~A()
};
```

27.6 Срезка (Slicing)

Так как публичное наследование реализует отношение "является можно инициализировать объекты предка объектами потомков, а также присваивать предкам потомков. Но не наоборот!

```
struct A {
  int x;
  void f();
};

struct B : public A { // B is A (but A is not B)
  int y;
  void g();
};

B b;
A a = B();
a = b;
b = a; // CE
```

Данная возможность называется "срезкой"(при присваивании используются только часть класса, относящаяся к базовому классу).

Наоборот нельзя, так как в B могут быть элементы, которых нет в A.

Стоит отметить, что если в базовом классе есть копирующий/перемещающий конструктор/присваивание, то при срезке будут использованы именно они.

При приватном и защищенном наследовании внешний код не имеет права использовать факт наличия связи между классами, поэтому срезка запрещена.

```
struct A { /* ... */ };
struct B : protected A { /* ... */ };

A a = B(); // CE
```

Однако существуют исключения:

1. внутри класса-наследника срезку делать можно (он "знает"про то, что он от чего-то унаследован)

```
struct B : protected A {
  void f() { A a = B(); } // Ok
};
```

2. друзья наследника тоже имеют право знать (и использовать) факт наследования.

Для задания конкретного поведения нужно определить конструктор A от B:

```
struct B; // forward declaration

struct A {
  std::string name = "A";
  A(const B& other);
  // ...
};

struct B : public A {
  // ...
};

A::A(const B& other) : name("A") { /* ... */ }

A aa = b; aa.name; // "A"
```

Если же мы хотим запретить срезку, то необходимо для данного конструктора использовать конструкция `= delete`

27.7 Затенение методов базового класса (shadowing)

Если в производном классе присутствует метод с тем же именем, что и метод в базовом классе, то это имя затеняет базовые методы с тем же именем (shadowing)

Есть несколько решений этой "проблемы"

1. Смириться.
2. Использовать полное имя метода (`b.A::f()`).
3. Ввести имена базового класса в область производного с помощью `using` :

```
struct A {
  void f();
  void f(double);
};

struct B : public A {
  using A::f; // A::f(), A::f(double) (partially impossible!)
  void f(int);
};

b.f(0); // Ok: B::f(int)
b.f(); // Ok: A::f()
b.f(0.0); // Ok: A::f(double)
```

27.8 Работа с производным классом через указатель или ссылку на базовый

Если класс B публично унаследован от класса A, можно (подобно срезке) присваивать указатель/ссылку на B указателю/ссылке на A, но не наоборот:

```
class B : public A { /* ... */ };

B b;
A* b_ptr = &b; // Ok
A& b_ref = b; // Ok

A a;
B* a_ptr = &a; // CE
B& a_ref = a; // CE
```

Указатель или ссылка ссылаются на тот же объект и работают с его данными, в отличие от срезки, где мы работаем с копией:

```
struct A {
  int x = 0;
  void f() { ++x; }
};

struct B : public A {
  int y = 0;
};

B b; // b.x == 0; b.y == 0
A a = b; // a.x == 0
a.f(); // a.x == 1, b.x == 0
a.y; // CE
A& b_ref = b;
b_ref.f(); // b.x = 1;
b_ref.y; // CE: type(b_ref) == A&
```

Компилятор выбирает версию метода, основываясь на статическом типе указателя/ссылки, не обращая внимания на реальный тип объекта, на который ссылаются:

```
struct A {
  void f() { std::cout << "A::f()\n"; }
};

struct B() : public A {
  void f() {std::cout << "B::f()\n"; }
};

B b;
A* b_ptr = &b;
b_ptr->f(); // A::f()
```

Для решения этой проблемы мы можем использовать виртуальные функции. Подробнее рассмотрим их в следующем блоке.

27.9 Виртуальные функции

Добавим `virtual` перед объявлением метода в классе A и, voila:

```
struct A {
  virtual void f() { std::cout << "A::f()\n"; }
};

struct B() : public A {
  void f() {std::cout << "B::f()\n"; }
};

B b;
A* b_ptr = &b;
b_ptr->f(); // B::f()
```

`virtual` перед объявлением метода говорит, что решение о том, какую версию метода выбрать, должно быть принято *во время исполнения программы* (**позднее связывание**), а не на *этапе компиляции* (**раннее связывание**)

```
struct A {
  virtual void f() { std::cout << "A::f()\n"; }
  void g() { std::cout << "A::g()\n"; }
};

struct B() : public A {
  void f() {std::cout << "B::f()\n"; } // automatically virtual!
  void g() { std::cout << "B::g()\n"; }
};

A* ptr = new B;
// at compile time, the call to A::g is "substituted"
ptr->g();
// the decision is postponed until the execution of the program
ptr->f();
```

Решение откладывать необходимо, так как реальный тип объекта может быть неизвестен:

```
int x;
std::cin >> x;
A* ptr = (x == 0 ? new B : new A);
// at compile time, the call to A::g is "substituted"
ptr->g();
// the decision is postponed until the execution of the program
ptr->f();
```

Как вы понимаете, позднее связывание более затратно, чем раннее связывание, за счет дополнительных действий во время исполнения программы (выяснение истинного типа объекта)

27.9.1 Вызов виртуальной функции из метода класса

"Виртуальность"работает и внутри методов:

```
struct A {
  void PrintName() const { std::cout << Name() << '\n'; }
  virtual const char* Name() const { return "A"; }
};

struct B : public A {
  const char* Name() const { return "B"; }
};

A* ptr = new B;
ptr->PrintName(); // B
```

Потому что `Name()` в `A::PrintName()` ⇔ `this->Name()`
А в конструкторах и деструкторах "виртуальность"не работает!

```
struct A {
  virtual void f() { std::cout << "A␣"; }
  A() { f(); }
  ~A() { f(); }
};

struct B : public A {
  void f() { std::cout << "B␣"; }
  B() { f(); }
  ~B() { f(); }
};

B b; // A B B A
```

27.10 Динамический полиморфизм

- Напоминание: **Полиморфизм** - свойство системы, позволяющее использовать различные реализации в рамках одного интерфейса. **Статический полиморфизм** - вид полиморфизма, при котором выбор реализации осуществляется на этапе компиляции (перегрузка функций, шаблоны, перегрузка операций и т.д)
- **Динамический полиморфизм** - вид полиморфизма, при котором выбор реализации осуществляется во время выполнения программы.
- Основной механизм реализации динамического полиморфизма в C++ - **виртуальные функции**.
- Поэтому наследование с применением виртуальных функций называют **полиморфным**

27.10.1 Пример применения

Простым применением является класс животных:

```
struct Animal {
  virtual void Voice() const { ... }
};

struct Cat : Animal {
  void Voice() const { std::cout << "Meow\n"; }
};

struct Dog : Animal {
  void Voice() const { std::cout << "Woof\n"; }
};

struct Fox : Animal {
  void Voice() const { std::cout << "???\n"; }
};

Animal* animals[10];
// ... (fill animals)
for (int i = 0; i < 10; ++i) {
  animals[i]->Voice();
}
```

27.11 Виртуальный деструктор

Рассмотрим сначала почему виртуальный деструктор вообще возникает. Возьмем следующие классы:

```
class Stack {
  // ...
};

class StackMax : public Stack {
  // ...
};

Stack* stack_ptr = new StackMax;
// ...
delete stack_ptr; // <-- UB
```

Следующие размышления приведут нас к проблеме:

- Тип `stack_ptr` — `Stack*` .
- Следовательно `delete` вызывает деструктор `Stack` , но не `StackMax` ! То есть, `StackMax` уничтожен некорректно, а это пропуск в чудесный мир *Undefined Behaviour*

На практике, скорее всего, вы столкнетесь с утечкой памяти (если производный класс выделял динамическую память или содержит поля с нетривиальными деструкторами).

Проблема в том, что версия деструктора выбиралась на этапе компиляции. Чтобы отложить это решение до момента фактического исполнения кода, необходио объявить деструктор виртуальным:

```
class Stack {
  // ...
  virtual ~Stack();
  // ...
};

class StackMax : public Stack {
  // ...
};

Stack* stack_ptr = new StackMax;
// ...
delete stack_ptr; // OK: destructor is selected at run time
```

Таким образом, для полиморфных классов необходимо писать виртуальный деструктор.

27.12 override

Для переопределения виртуальной функции в производном классе необходим (почти) в точности сохранить тип функции. В противном случае компилятор будет считать это определением нового метода (не связанного с предыдущим).

Чтобы убедиться, что вы действительно переопределяете какую-либо виртуальную функцию базового класса (не ошиблись в определении функции), а не создаете новую, можно написать `override` после типа функции.

```
struct A {
  virtual void f(int);
  virtual void g() const;
};

struct B : public A {
  void f(long) override;
  void g() override;
};
```

Теперь компилятор выдаст CE в случае несоответствия типов функций:

```
error: 'void␣B::f(long␣int)' marked 'override', but does not override
error: 'void␣B::g()' marked 'override', but does not override
```

27.12.1 Ковариантные возвращаемые типы

Тип виртуальной функции и ее переопределения должны быть равны с точностью до *ковариантных* возвращаемых типов.

Типы называются ковариантными, если они являются указателями или ссылками и ссылаются на родственные классы (предок-потомок).

Короче говоря, если виртуальная функция возвращает указатель/ссылку на класс, то при переопределении можно вернуть указатель/ссылку на производный класс

```
struct A {
// ...
virtual A* Clone() const { return new A(*this); }
};

struct B : public A {
  // ...
  B* Clone() const override { return new B(*this); }
  // Ok: B* and A* are covariant
};
```

27.13 final

Чтобы запретить дальнейшее переопределение виртуального метода можно пометить его словом `final` :

```
struct A {
  virtual void f();
};

struct B : public A {
  void f() final;
};

struct C : public B {
  void f() override; // CE
};
```

27.13.1 Использование final

- Финальной можно пометить только виртуальную функцию:

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

struct B : public A {
  void f() final; // CE: f is not virtual!
};
```

- Можно объявить виртуальную функцию и сразу сделать ее финальной:

```
struct A {
  virtual void f() final;
};
```

Это имеет смысл, если хочется запретить наследникам определять функции с тем же именем и типом.

- `final` можно использовать и при определении класса, чтобы запретить от него наследоваться:

```
struct A {
  // ...
};

struct B final : public A {
  // ...
};

struct C : public B {}; // CE
```

27.14 Чисто виртуальные функции и абстрактные классы

Допустим, вы решили написать свой мессенджер. Необходимо поддержать кучу видов сообщений (текстовые, видео, стикеры, голосовые)

```
class Message {
  // ...
  void Send(Chat* chat_ptr) const;
  virtual void Display() const; // must be overridden for all types
};

class TextMessage : public Message {
  // ...
  void Display() const override; // text message display
};

class StickerMessage : public Message {
  // ...
  void Display() const override; // sticker display
};

// ... other classes
```

Для решения проблемы необходимо указать, что `Message` не является полноценным типом, а представляет из себя лишь интерфейс, то есть набор методов характерных для каждого сообщения.

Для этого необходимо объявить методы интерфеса *чисто виртуальными*:

```
class IMessage {
  // ...
  virtual void Display() const = 0;
};
```

27.14.1 Правила работы

- Чисто виртуальные функции (за исключением чисто виртуальных деструкторов) можно оставлять без реализации

```
class IMessage {
  // ...
  virtual void Display() const = 0; // Ok
  virtual ~IMessage() = 0; // CE: implementation needed
};
```

- Чисто виртуальные функции могут быть реализованы только вне класса:

```
class IMessage {
  // ...
};

IMessage::~IMessage() { MessageCleanup(); }
```

27.15 Абстрактный класс

Абстрактный класс — класс с хотя бы одним чисто виртуальным методом.

```
class IMessage {
  // ...
  virtual void Display() const = 0; // Ok
};
```

27.15.1 Правила работы

- Нельзя создавать объекты абстрактного класса
- Можно создавать ссылки и указатели на абстрактный класс
- Если наследник не переопределяет (реализует) чисто виртуальную функцию, то он сам становится абстрактным:

```
class Message {
  // ...
  virtual void Display() const = 0;
};

class TextMessage : public Message {
  // ...
  void Display() const override { // ... }
};

class StickerMessage : public Message {
  // ...
  // Display is not overridden => the class is abstract
};

TextMessage text; // Ok
StickerMessage sticker; // CE
```

- Вызов чисто виртуального метода в конструкторе/деструкторе абстрактного класса приводит к undefined behaviour

```
class IMessage {
  // ...
  virtual void Display() const = 0;

  IMessage() {
    // ...
    Display(); // UB
  }
};
```

- Абстрактные классы используются для определения интерфейсов для создания семейства классов с одинаковыми свойствами и методами

28 Move-семантика

28.1 Категории значений

Категория значения - вторая характеристика выражения в C++ (первая - тип).

Главный вопрос, на который отвечает категория значения:

материален ли результат этого выражения, то есть существует ли он в виде объекта в памяти или нет? (идентичность)

28.1.1 lvalue

lvalue - категория значений, которая обладает идентичностью (но не перемещаема).

Неформально: к *lvalue* относится все, у чего есть постоянное место в памяти (прописка). Критерий - у выражения можно получить адрес.

- Переменная - всегда *lvalue*!
- Результат функции/операции, возвращающей ссылку
- Строковый литерал (относится к массиву, который содержит символы)

```
x; // lvalue, &x - valid operation
++x; // lvalue, &++x - valid operation
"abc"; // lvalue, &"abc" - valid operation
x + 1; // not lvalue, &(x + 1) - invalid operation
```

28.1.2 rvalue

rvalue — перемещаемая категория значений. Делится на *prvalue* (pure rvalue) - значение без идентичности и *xvalue* (expired value) - значение с идентичностью.

Неформально: к *rvalue* относится все, что не относится к *lvalue* (временные значения).

- Литералы (кроме строкового)
- Результат функции/операции, возвращающей значение
- `this`
- Значение типа перечисления (`enum`)
- Параметр шаблона не являющийся типом (если это не ссылка)

28.2 Виды ссылок

28.2.1 lvalue ссылки

lvalue-ссылка - это ссылка, которая может связываться с результатом lvalue выражения. Эта ссылка становится псевдонимом объекта, на который она ссылается

```
int& f();
int x = 0;

int& rx = x; // lvalue reference on x
const int& crx = x; // const lvalue reference on x
int& ref = f(); // lvalue reference on result f()
int& rx2 = ++x; // lvalue reference on x

int& y = 0; // CE
int& z = x++; // CE
```

Исключение - константная lvalue ссылка (продлевает жизнь временного объекта)

```
// 11 ends up in a memory area named tmp
const int& tmp = 11;
```

28.2.2 rvalue ссылки (C++11)

rvalue-ссылка — это ссылка, которая может связываться с результатом *rvalue* выражения. Эта ссылка продлевает жизнь объекта и в отличие от константных lvalue ссылок позволяет изменять объект

```
int f();

int x = 0;
int&& rry = 11; // rvalue reference, initialized to 11
rry = 12; // Ok
int&& rrz = f(); // Ok
int&& rrt = x++; // Ok

int&& y = x; // CE
int&& z = ++x; // CE
int& ref = rry; // Ok: variable is ALWAYS an lvalue
```

28.3 Перегрузка функций по виду ссылки

По виду ссылки можно перегружать функции. Тогда результат lvalue выражений будет вызывать версию с левой ссылкой, а результат rvalue выражений - с правой

Например, достаточно реализовать функцию принимающую аргумент по rvalue-ссылке. Тогда именно она будет выигрывать перегрузку в случае временных объектов.

```
template <class T>
const T* AddressOf(const T& value) {
  return &value;
}

template <class T>
const T* AddressOf(const T&&) = delete;
// ...
int x = 0;
auto array = new int[10];

AddressOf(x); // Ok
AddressOf(array[5]); // Ok
AddressOf(10); // CE: use of deleted function
```

Аналогично можно поступать не только с функциями, но и с методами. В том числе и с конструкторами.

28.4 Перемещение

Вспомним конструктор копирования и копирующее присваивание стека

```
Stack<T>::Stack(const Stack<T>& other) :
  buffer_(new T[other.capacity_]) ,
  size_(other.size_) {
  for (size_t i = 0; i < size_; ++i) { // O(N)
    buffer_[i] = other.buffer_[i];
  }
}

Stack<T>& Stack<T>::operator=(const Stack<T>& other) {
  if (this != &other) {
    // ... delete[] buffer_;
    // ... fill new buffer O(N)
  }
  return *this;
}
```

Таким образом, мы полчим с вами следующую проблему:

```
Stack<int> stack;
// ...

auto lcopy = stack; // copy stack: O(N)
// before C++17: object creation and copy: O(N) + O(N)
auto rcopy = Stack<int>(100);

lcopy = stack; // copy lcopy: O(N)
// creating and copying a temporary object: O(N) + O(N)
rcopy = Stack<int>(100);
```

Для того чтобы избежать ненужного копирования, давайте напишем специальные версии конструктора и присваивания, которые принимают rvalue-ссылки (то есть работают со временными значениями)

28.4.1 Конструктор перемещения и перемещающее присваивание

Для того, чтобы лишних копирований не происходило, давайте реализуем контруктор и присваивание следущим образом:

```
Stack(Stack&& other) noexcept :
  buffer_(other.buffer_), size_(other.size_) {
  other.buffer_ = nullptr;
  other.size_ = 0;
}

Stack& operator=(Stack&& other) noexcept {
  if (this != &other) {
    delete[] buffer_;
    buffer_ = other.buffer_;
    size_ = other.size_;
    other.buffer_ = nullptr;
    other.size_ = 0;
  }
  return *this;
}

// before C++17: create and move: O(N) + O(1)
auto rcopy = Stack(100);
rcopy = Stack(100); // create and move: O(N) + O(1)
```

Данные методы называются перемещающим конструктором и перемещающим присваиванием соответственно.

- Если вы не определили своего копирования, присваивания и деструктора (ничего из этого), то компилятор предоставит вам свой конструктор перемещения. Он вызовет конструктор перемещения для каждого из полей.
- Аналогично для перемещающего присваивания
- Для этих методов можно писать `= default`
- По причинам, которые обсудим позже, они должны быть `noexcept` !

28.4.2 Правило пяти (C++11)

В современном C++ правило трех эволюционировало до правила пяти:

Если класс требует реализации хотя бы одного метода из списка:

1. Конструктор копирования
2. Конструктор перемещения
3. Копирующее присваивание
4. Перемещающее присваивание
5. Деструктор

то требуется реализовать их все

Как и "правило трех"это не правило языка, но для корректной работы программ следовать ему обязательно.

28.5 std::move

Если мы с вами рассмотрим простую реализацию `Swap`

```
template <class T> void Swap(T& x, T& y) {
  T tmp = x;
  x = y;
  y = tmp;
}
```

то в случае, например, со стеком, мы получим очень долгие копирования:

```
Stack a(100'000'000, 0);
Stack b(100'000'000, 11);

Swap(a, b); // 3 copies
```

Поэтому хотелось бы не копировать объекты, а именно перемещать. Но переменная это же всегда lvalue! Поэтому здесь всегда будет вызываться конструктор копирования.

Необходим способ заставить компилятор воспринимать выражение справа как rvalue.

В языке C++ есть преобразование из lvalue в rvalue.

Это преобразование осуществляется с помощью функции `std::move` .

`std::move` **не меняет** состояния объекта. Он лишь просит объект ненадолого *притвориться* временным (хотя таковым он являться не будет).

Для результата `std::move` есть специальная категория - *xvalue* (частный случай rvalue - перемещаемый и с идентичностью)

```
int x = 11;
std::move(x); // nothing will happen to x!

int& rx = std::move(x); // CE: right rvalue!
int y = std::move(x); // Ok
int&& rrx = std::move(x); // Ok: right rvalue

rrx = -1; // rrx is related to x value!
std::cout << x << ' ' << rrx; // -1 -1
```

У базовых типов нет семантики перемещения, поэтому эти примеры скучные

Рассмторим стек:

```
Stack x = y; // copy constructor
y = x; // copy assignment

std::move(y); // Nothing happens to y!
Stack z = std::move(y); // move constructor
y = std::move(x); // move assignment
```

В последнем случае не создано ни одного нового буфера!

Но x теперь пустой.

Пустым его сделал не `std::move` , а перемещающий оператор присваивания, реализованный нами!

Таким образом, `Swap` здорового программиста выглядит так:

```
template <class T> void Swap(T& x, T& y) {
  T tmp = std::move(x); // move the contents of x to tmp
  x = std::move(y); // move contents of tmp to x
  y = std::move(tmp); // move contents of tmp to y
}
```

28.6 Forwarding reference (универсальная ссылка)

28.6.1 Сворачивание ссылок

В процессе подстановки шаблонных параметров может возникнуть "ссылка на ссылку". Несмотря на то, что они запрещены стандартом, в этом случае действуют специальные правила.

```
template <class T>
void f(T& x, T&& y);
f<int&>(...);
// T = int&, type(x) == int&, type(y) == int&
f<int&&>(...);
// T = int&&, type(x) == int&, type(y) == int&&
```

Правила сворачивания ссылок при подстановке шаблонных параметров:

- type& & == type&
- type& && == type&
- type&& & == type&
- type&& && == type&&

28.6.2 Forwarding reference

Универсальная ссылка - это ссылка одного из следующих двух видов:

```
//T&& - Forwarding reference
template <class T>
void Function(T&& x);

//auto&& - Forwarding reference
auto&& x = /* ... */;
```

То есть универсальная ссылка имеет вид обычной (без модификаторов) rvalue- ссылки (но ей не является!) и применяется только к шаблонным параметрам функции или к объявлениям `auto` .

```
template <class T>
void Function(const T&& x); // <-- not forwarding reference! (const)

const auto&& x = /* ... */; // <-- not forwarding reference! (const)
template <class T>
class Stack {
  // ...
  void Push(T&& value); // <-- not forwarding reference!
  // (is not a template function parameter)

  template <class U>
  void Push(U&& value); // <-- forwarding reference
};
```

28.6.3 Правила вывода для универсальных ссылок

- `cv` квалификаторы не отбрасываются.
- При передаче lvalue в качестве аргумента тип T выводится как lvalue-ссылка.
- При передаче rvalue в качестве аргумента тип T выводится как нессылочный.

28.6.4 Универсальные ссылки: примеры

```
template <class T>
void Function(T&& arg);

int x = 0;
const int cx = x;
int&& rx = 0;
const int&& crx = 0;

Function(x); // [T = int&], type(arg) == int&
Function(cx); // [T = const int&], type(arg) == const int&
Function(rx); // [T = int&], type(arg) == int&
Function(crx); // [T = const int&], type(arg) == const int&
Function(0); // [T = int], type(arg) == int&&
Function(std::move(x)); // [T = int], type(arg) == int&&
Function(std::move(cx)); // [T = const int], type(arg) == const int&&
Function<int&>(x); // [T = int&], type(arg) == int&
Function<int&&>(0); // [T = int&&], type(arg) == int&&
Function<int>(0); // [T = int], type(arg) == int&&
```

28.6.5 Проблемы

Рассмотрим реализацию `RunningTime`

```cpp
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
  const auto start = std::clock();
  func(arg);
  return std::clock() - start;
}

void FunctionR(int&&);

FunctionR(x) /* CE */; RunningTime(FunctionR, x) /* CE */;
FunctionR(0) /* Ok */; RunningTime(FunctionR, 0) /* CE */;

void FunctionL(int&);

FunctionL(0) /* CE */; RunningTime(FunctionL, 0) /* Ok */;
FunctionR(0) /* Ok */; RunningTime(FunctionR, 0) /* CE */;
```

Таким образом, возникают следующие проблемы:

- Наблюдается ассиметричность: хотелось бы, чтобы там, где было CE, оставалось CE, и аналогично для Ok
- Невозможно использовать `FunctionR` !

Дело в том, что внутри `RunningTime` arg всегда является lvalue!
Хотелось бы, чтобы:

- Если в arg передали lvalue, то вызывается func(arg)
- Если в arg передали rvalue, то вызывается func(std::move(arg))

И это возможно, так как универсальные ссылки по-разному выводят Arg при передаче lvalue и rvalue.

28.6.6 std::forward

В C++ есть специальная функция, которая осуществляет "условный std::move "(делает `std::move` , если аргумент был принят как rvalue, и не делает, если был принят как lvalue).
`std::forward<Arg>(arg) <=> if (Arg == type&) arg; else std::move(arg)`
Теперь все работает корректно:

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
  const auto start = std::clock();
  func(std::forward<Arg>(arg));
  return std::clock() - start;
}

void FunctionR(int&&);

FunctionR(x) /* CE */; RunningTime(FunctionR, x) /* CE */;
FunctionR(0) /* Ok */; RunningTime(FunctionR, 0) /* Ok */;

void FunctionL(int&);

FunctionL(x) /* Ok */; RunningTime(FunctionL, x) /* Ok */;
FunctionL(0) /* CE */; RunningTime(FunctionL, 0) /* CE */;
```

28.7 Copy Elision

Copy Elision - оптимизация, позволяющая избежать копирования/перемещения объектов, при передаче временных (иногда локальных) объектов по значению.

При выполнении данной оптимизации конструкторы копирования и перемещения игнорируются, даже, если они имеют "побочные эффекты"или объявлены как удаленные или приватные.

```
class A {
  A(const A&) = delete;
  A(A&&) = delete;
 public:
  A() = default;
};

A f() { return {}; }
A a = A(); // Ok: only the default constructor is called
A b = f(); // Ok: only the default constructor is called
```

Основные контексты проявления copy elision:

- Инициализация объекта с помощью prvalue выражения того же типа (с точностью до const и volatile)

  `A a = A(); A b(A());`

- *Return Value Optimization (RVO)*: Возврат из функции prvalue выражения того же типа, что и тип возвращаемого значения (с точностью до cv)

  ```
  A f() { return {}; }
  ```

- *Named Return Value Optimization (NRVO)*: То же, что и RVO, но в return выражении стоит локальная переменная не являющаяся аргументом функции.

```
A f() { A a; return a;}
```

28.7.1 Не обязательный copy elision (до C++17)

- До C++11 (C++98, C++03) copy elision не был частью стандарта C++. Компиляторы реализовывали эту оптимизацию в обход правил языка.
- В C++11 copy elision был включен в стандарт в качестве оптимизации, которую могут реализовывать компиляторы, но не обязаны (non-mandatory copy elision).
- Чтобы отключить эти оптимизации необходимо было передать дополнительный флаг компиляции `-fno-elide-constructors`

28.7.2 Обязательный copy elision (C++17)

- В C++17 copy elision стал частью языка и теперь он гарантируется в некоторых ситуациях (mandatory copy elision)
- Copy elision происходит при инициализации объекта с помощью prvalue (1 пункт) и RVO (2 пункт). На это даже не может повлиять флаг `-fno-elide- constructors`
- В остальных контекстах (в том числе NRVO) стандарт оставляет все на совести компилятора (ничего не гарантируется).

29 VariadicTemplates

Предположим, что мы хотим написать функция, которая сможет принимать произвольное число аргументов. Рассмотрим возможности, которые нам могут в этом помочь.

29.0.1 Variadic functions (C-style)

Для обозначения функции, принимающей произвольное число аргументов, в языке C использовался специальный аргумент - `...` (elipsis)

```cpp
#include <cstdarg>

void PrintInts(std::ostream& os, int n, ...) {
  std::va_list args; va_start(args, n);
  for (int i = 0; i < n; ++i) {
    os << va_arg(args, int) << (i == n - 1 ? "" : "␣");
  }
  va_end(args);
}

float SumDoubles(int n, ...) {
  std::va_list args; va_start(args, n);
  auto res = 0.0;
  for (int i = 0; i < n; ++i) { res += va_arg(args, double); }
  va_end(args);
  return res;
}
```

29.1 Variadic templates (C++11)

C++ предоставляет более безопасный интерфейс для работы с функциями с переменным числом аргументов

Синтаксис:

```cpp
template <class... Args> // or template <typename... Args>
void Print(std::ostream& os, Args... args);

template <class... Args>
auto Sum(Args... args);
```

- `Args` - пакет параметров-типов (type template parameter pack)
- `args` - пакет параметров функции (function parameter pack)

```cpp
Print(std::cout, 0, "a");
// Args = [int, const char*], args = [0, "a"]

Sum(0, 1.0, 1u);
// Args = [int, double, unsigned], args = [0, 1.0, 1u]
```

29.1.1 Правила работы

- Пакеты параметров можно использовать и в шаблонах классов, но только в качестве последнего параметра.

```
template <class T, class... Args> class MyClass { /* ... */ };
// template <class... Args, class T> class MyClass; - error

MyClass<int> x; // T = int, Args = []
MyClass<int, char, float> y; // T = int, Args = [char, float]
```

- В случае шаблонов функций пакеты параметров могут идти как в начале, так и в конце. Но лучше всегда писать в конце, иначе возможны проблемы

```
template <class... Args, class T>
void Function(T x, Args... args); // ok

template <class... Args, class T>
void Function(Args... args, T x); // ok, but can't use ...
```

29.1.2 Примеры использования

Рассмарим некоторые примеры и ошибки, при использовании Variadic templates:

```
template <class T, class... Args>
void Function(T x, Args... args);

Function(0, 0.0); // ok: T=int, Args=[double,]
Function<int, int, int>(0, 0.0, 1); // ok: T=int, Args=[int, int]
```

```
template <class... Args, class T>
void Function(T x, Args... args);

Function(0, 0.0); // ok: T=int, Args=[double,]
Function<int, int, int>(0, 0.0, 1); // Fail: Args=[int, int, int], T=?
```

```
template <class T, class... Args>
void Function(Args... args, T x);

Function(0, 0.0); // Fail: Args=[int, double], T=?
Function<int, int, int>(0, 0.0, 1); // ok: T=int, Args=[int, int]
```

```
template <class... Args, class T>
void Function(Args... args, T x);

Function(0, 0.0); // Fail
Function<int, int, int>(0, 0.0, 1); // Fail
```

29.2 Использование Args...

29.2.1 Оператор sizeof...

`sizeof...` применяется к пакету параметров и возвращает количество элементов в пакете (именно количество, а не размер в байтах).

```
template <class... Args>
void Function(const Args&... args) {
  // ...
  std::cout << sizeof...(Args) << ' ';
  std::cout << sizeof...(args) << ' ';
  // ...
}

Function(1, 2, 3); // 3 3
```

29.2.2 Распаковка пакета

Если у вас уже есть функция с переменным числом аргументов, то в нее можно передать пакет параметров. Для этого в месте использования необходимо к его имени добавить `...` .

```
template <class... Args>
void PrintTitle(std::ostream& os, std::string_view title, Args... args)
  os << title << '\n';
  Print(os, args...); // <=> Print(os, args[0], ..., args[n-1]);
}
```

Также при распаковке к пакету можно применить некоторый паттерн, который будет применен к каждому элементу пакета (`pattern(args)...`):

```
template <class... Args>
auto SumSquares(Args... args) {
  return Sum(args * args...);
  // <=> Sum(args[0] * args[0], ..., args[n-1] * args[n-1])
}
```

29.2.3 Метод откусывания

Нельзя просто так взять и обратиться к элементу пакета параметров.

Общий способ работы - обрабатывать элементы по одному и использовать рекурсию.

```cpp
template <class Head> // end of recursion (Args = [])
void Print(std::ostream& os, Head head) {
  os << head;
}

template <class Head, class... Args> // "bit off" Head
void Print(std::ostream& os, Head head, Args... args) {
  os << head << ' ';
  Print(os, args...); // passing arguments from args further
}

Print(std::cout, 1, "is greater than", 0.0); // "1 is greater than 0"
```

Еще одним примером является:

```cpp
template <class Head> // end of recursion (Args = [])
auto Sum(Head head) {
  return head;
}

template <class Head, class... Args> // "bit off" Head
auto Sum(Head head, Args... args) {
  return head + Sum(args...); // passing arguments from args further
}

Sum(1, 1.5, 0l); // 2.5
```

29.2.4 Fold expression (C++17)

Fold expression (выражение свертки) - более удобный интерфейс использования пакетов параметров, позволяющий в большинстве ситуаций обойтись без рекурсии

Синтаксис:

```
op - some binary operation

(pack op ...) <=> (a[0] op (... op (a[n-2] op a[n-1])))
(... op pack) <=> (((a[0] op a[1]) op ...) op a[n-1])
(pack op ... op init) <=> (a[0] op (... op (a[n-1] op init)))
(init op ... op pack) <=> (((init op a[0]) op ...) op a[n-1])
```

Примеры использования:

```cpp
template <class... Args>
void Print(std::ostream& os, const Args&... args) {
  (os << ... << args); // os << args[0] << args[1] ... << args[n-1];
}
```

```cpp
template <class... Args>
auto Sum(const Args&... args) {
  return (args + ...); // args[0] + args[1] + ... + args[n-1]
  // or return ... + args;
}
```

30 Разговор про new

30.1 new

Рассмотрим следующие примеры:

```cpp
new T; // 1
new T(x, y); // 2
new T[n]; // 3
```

При выделении памяти происходит следующее:

1. Выделяет память достаточную для хранения объекта типа T. Создает объект по умолчанию в выделенном месте.
2. Выделяет память достаточную для хранения объекта типа T. Создает объект с параметрами x и y в выделенном месте.
3. Выделяет память достаточную для хранения n объектов типа T. Создает n объектов по умолчанию.

Но хотелось бы выделять память без вызова конструкторов и создать объект в нужном месте, без выделения памяти.

30.2 Выделение памяти

В наследство от языка C нам досталась функция malloc . Так как в C нет конструкторов, то и malloc о них ничего не знает!

```cpp
// malloc takes a number of bytes, returns void*
auto single = std::malloc(sizeof(T));
auto array = std::malloc(sizeof(T) * n);

std::free(single);
std::free(array);
```

В качестве альтернативы в C++ можно использовать массивы char:

```cpp
auto single = static_cast<void*>(new char[sizeof(T)]);
auto array = static_cast<void*>(new char[sizeof(T) * n]);

// but it will have to be cleared as char arrays
delete[] static_cast<char*>(single);
delete[] static_cast<char*>(array);
```

30.3 operator new

Социально одобряемым способом выделения и очищения памяти в C++ является использование функций `operator new` и `operator delete` :

```
// operator new takes a number of bytes, returns void*
auto single = operator new(sizeof(T));
auto array = operator new(sizeof(T) * n);

// operator delete takes void*, no explicit cast required
operator delete(single);
operator delete(array);
```

Именно ее вызывает new при выделении памяти под объект:

```
auto ptr = new T; // <=> operator new + constructor call
delete ptr; // <=> destructor call + operator delete
```

30.4 operator new[]

Также существуют функции `operator new[]` и `operator delete[]` . Они вызываются при использовании операций `new[]` и `delete[]` .

```
auto array = new T[n]; // operator new[] + n constructors
delete[] array; // n destructors + operator delete[]
```

Версии с `[]` сохраняют дополнительную информацию о размере массива. Иначе откуда бы `delete[]` узнал, сколько деструкторов звать?

30.5 Формы operator new

У `operator new` есть несколько форм, наиболее употребимые из них:

```
// Called with normal new, allocates count bytes
void* operator new(size_t count);

// Called when new(std::nothrow) T(...), allocates count bytes,
// returns nullptr if out of memory, instead of throwing an exception
void* operator new(size_t count, std::nothrow_t)

// Called on placement new (this is a gun that will still fire)
// Doesn't allocate memory! Just returns ptr
void* operator new(size_t count, void* ptr);
```

В общем случае, при использовании `new(args...) T(...);` , вызывается функция `operator new(count, args...)` , где *count* - размер необходимой памяти.

30.6 Перегрузка и замена operator new

`operator new` можно перегрузить или даже заменить встроенные версии!

```
void* operator new(size_t count) { // replacement
  std::cout << "Trying␣to␣allocate␣" << count << "bytes!\n";

  // do not forget to replace operator delete
  return std::malloc(count);
}

void* operator new(size_t count, bool always_throw) { // overloading
  if (always_throw) {
    throw std::bad_alloc();
  }
  return ::operator new(count);
}

struct A {
  // never allocate on heap
  static void* operator new(size_t count) = delete;
  static void* operator new[](size_t count) = delete;
};
```

30.7 placement new

Рассмотрим следующую форму `new`, которая называется *placement new*.

```
// Creating an object on the stack
char object[sizeof(T)];
// called by default at object address
auto ptr = new(object) T;

// The compiler does not know that the object array contains an object
ptr->~T(); // so we call the destructor manually

// Creating an object on the heap
auto ptr = static_cast<T*>(operator new(sizeof(T)));
//the constructor is called with parameters x, y at the address object
new(ptr) T(x, y);

// delete ptr; most likely it will work, but formally UB
ptr->~T();
operator delete(ptr);
```

30.8 Использование в std::vector

Теперь рассмотрим, как устроены size и capacity:

```
buffer = static_cast<T*>(operator new(sizeof(T) * capacity));
for (int i = 0; i < size; ++i) {
  new(buffer + i) T;
}
```

В деструкторе отдельно удаляем объекты и очищаем память:

```
for (int i = 0; i < size; ++i) {
  (buffer + i)->~T();
}
operator delete(buffer);
```

1. clear не просто выставляет size=0, но и вызывает size деструкторов
2. resize не просто обновляет size, но и вызывает деструкторы/конструкторы
3. reserve выделяет память нужного размера и переносит size объектов

31 Исключения

31.1 Возможные способы обработки ошибок

1. Никак

   ```
   template <class T>
   T Divide(T x, T y) {
     if (y == 0) {}
     return x / y;
   }
   ```

   Очевидно, не всегда хороший способ (но многие пользуются).

2. Вернуть специальное значение.

   ```
   template <class T>
   T Divide(T x, T y) {
     if (y == 0) { return kSpecialValue<T>; }
     return x / y;
   }
   ```

   Но как отличить специальное значение от результата? Что, например, вернуть при делении int 'ов?

3. Вернуть код ошибки, а значение записать в выходной аргумент функции.

```
template <class T>
error_t Divide(T x, T y, T* out) {
  if (y == 0) { return 1; }
  *out = x / y;
  return 0;
}
```

4. Записывать код ошибки в специальную переменную

```
template <class T>
T Divide(T x, T y) {
  if (y == 0) { errno = 1; }
  errno = 0;
  return x / y;
}
```

Получаем весь пакет проблем, связанных с глобальными переменными.

31.2 Обработка ошибок в C

В языке C чаще всего используется способ с возвратом кода ошибки.

```
error_t f() {
  int err = g();
  if (err) { /* do something */ return err;}
  err = h();
  if (err) { /* do something */ return err;}
  return 0;
}
```

- Код раздувается из-за большого числа проверок и ветвлений.
- Для получения результата необходимо передавать указатель/ссылку.
- Легко проигнорировать возвращаемое значение.

31.3 Обработка ошибок в C++

В C++ используется механизм исключений.

Исключение - объект, который генерируется при возникновении исключительной ситуации (ошибки). По идее, должен содержать необходимую информацию о природе ошибки.

Генерация ошибки происходит с помощью оператора `throw` :

```
template <class T>
T Divide(T x, T y) {
  if (y == kZero<T>) { throw 1; } // Exception of type int, value 1
  return x / y;
}
```

31.3.1 Оператор throw

После выполнения `throw` работа функции прекращается, для объектов на стеке вызываются деструкторы. Та же участь постигает остальные функции (раскручивание стека), то есть всю последовательность вызовов, приведшую к ошибке.

```
void Cancer() { throw 1; std::cout << "Cancer()::here\n"; }
void Smoking() { Cancer(); std::cout << "Smoking()::here\n"; }
int main() { Smoking(); std::cout << "main()::here\n"; return 0; }
```

В данном примере main вызывает Smoking , Smoking вызывает Cancer , Cancer генерирует исключение 1 и тут же завершает работу. Smoking не обрабатывает это исключение, поэтому тоже завершает работу.

Аналогично погибает main .

Подробнее про `throw <obj>;`

Объект исключения хранится в специальном месте памяти и создается путем копирования или перемещения переданного объекта.

```
A a;
throw a; // Exception A is thrown as a copy of object a
throw std::move(a); // Exception object is created by move
```

При бросании prvalue происходит copy elision (C++17):

```
throw A(); // the default constructor is called, without copy or move
```

31.4 try-catch блок

Можно ли как-то перехватить ошибку и отложить (возможно, отменить) падение программы?

Да, для этого нужно воспользоваться конструкцией `try-catch`

```
template <class T>
T Divide(T x, T y) {
  f (y == 0) { throw 1; }
  return x / y;
}

int main() {
  try {
    Divide(1, 0);
  } catch (int err) {
    std::cout << "DivisionError:␣error␣code␣" << err << '\n';
  }
  return 0;
}
```

Возникающее в блоке try исключение может быть поймано в catch блоке соответствующего типа.

По завершении блока catch исключение считается успешно обработанным и выполнение программы продолжается в нормальном режиме.

Одному блоку try может соответствовать несколько блоков catch

```
int main() {
  try { Divide(1, 0); // throw an int
  } catch (double err) {
    std::cout << "double\n";
  } catch (int err) {
    std::cout << "int\n";
  }
  return 0;
}
```

В этом случае отработает только 1 блок, соответствующий типу брошенного исключения.

Если нужный блок catch не найден, то исключение поймано не будет. Вызов функции завершится и исключение полетит дальше.

```
int main() {
  try { Divide(1, 0); // throws an int
  } catch (double err) {
    std::cout << "double\n";
  } catch (const char* err) {
    std::cout << "const␣char*\n";
  }
  return 0;
}
```

Исключение не обработано:

```
terminate called after throwing an instance of 'int'
```

31.4.1 Выбор блока catch

Блок catch выбирается только по точному соответствию. То есть приведений типов НЕ происходит.

Из этого правила есть 2 исключения:

1. `void*` может поймать любой указатель

   ```
   void f() {
     int x;
     try { throw &x; }
     catch (void* ptr) { std::cout << "catched\n"; }
   }
   ```

2. Приведения по иерархии наследования вверх (к родителям) работают.

```
class B : public A {};

void f() {
  try { throw B(); }
  catch (A a) { std::cout << "catched\n"; }
}
```

Срабатывает всегда первый подходящий catch.

```
terminate called after throwing an instance of 'int'
```

31.4.2 Ловля исключений по ссылке

Если ловить исключение по значению, то исходное исключение скопируется.

Чтобы избежать копирования можно ловить исключения по ссылке (или по константной ссылке).

```
void f() {
  try { throw std::vector<int>(1 000 000); }
  catch (const std::vector<int>& v) {}
}
```

Более того, при ловле по ссылке становится доступно полиморфное поведение (позже рассмотрим, как это помогает)

31.4.3 catch(...)

Существует особый синтаксис catch , позволяющий поймать любое исключение.

```
int main() {
  try {
    f();
    g();
  } catch (...) {
    std::cout << "catched\n";
  }
}
```

Что бы и откуда бы (из `f()` или `g()`) не было брошено, оно будет поймано в `catch(...)` .

Но в таком блоке нельзя определить тип исключения и поработать с объектом исключения.

Не нужно ловить исключения просто для того, чтобы поймать.

Если непонятно как решить проблему, лучше не трогать его (пусть летит дальше).

Но иногда необходимо перехватить исключение, чтобы, например, освободить выделенные ресурсы (естественно, при раскручивании стека компилятор не догадается вызвать `delete`):

```
void f() {
  auto ptr = new int(11);
  try {
    g(ptr); // potentially throws an exception of type A
  } catch (A& a) {
    delete ptr;
    ptr = nullptr;
  }
  // ...
  delete ptr;
}
```

31.4.4 throw;

Существует особая форма оператора `throw` без аргумента - `throw;` .

Она дословно означает: "снова бросить пойманное исключение". При этом копии исключения не создается - будет "лететь"тот же объект, что и раньше.

```
void f() {
  auto ptr = new int(11);
  try {
    g(ptr); // potentially throws an exception of type A
  } catch (A& a) {
    delete ptr;
    throw; // throw old exception
  }
  // ...
  delete ptr;
}
```

31.5 Статическая спецификация исключений (C++11)

31.5.1 Спецификатор noexcept (C++11)

Чтобы пообещать компилятору, что функция не будет бросать исключений можно воспользоваться спецификатором noexcept

```
void f() noexcept {
  // ...
}
```

Если обещание будет нарушено, и исключение вылетит из noexcept функции, то программа завершится аварийно (без возможности перехватить исключение)

То есть в noexcept функциях нужно использовать только noexcept операции, либо перехватывать все исключения (так себе вариант):

```
void f() noexcept {
  try {
    g(); // potentially throwing exceptions
  } catch (...) {
    // solve the problem
  }
}
```

Компиляторы не выдают предупреждения при небезопасных вызовах в noexcept функциях, максимум - замечание.

31.5.2 Условный спецификатор noexcept (C++11)

noexcept может быть условным:

```
void f() noexcept; // <=> void f() noexcept(true);
void g(); // <=> void g() noexcept(false);
```

Это может быть использовано для маркирования функций, спецификатор которых зависит от некоторого условия (проверяемого на этапе компиляции):

```
template <class T>
void h(T x) noexcept(sizeof(T) > 1);
```

31.5.3 Операция noexcept

Ключевое слово `noexcept` еще служит для обозначения операции, которая определяет является ли выражение `noexcept` или нет:

```
void f() noexcept;
void g();
noexcept(f()); // true
noexcept(g()); // false

std::vector<int> v;
//true - integer division does not throw exceptions!
noexcept(1 / 0);
// false - push_back can throw out_of_memory
noexcept(v.push_back(1 / 0));
```

Важно понимать, что `noexcept` (ровно как и `sizeof`) **не вычисляет** результат выражения, а просто анализирует его на предмет возможности/невозможности получить исключение.

31.5.4 Финальный пример

Комбинируя условный спецификатор `noexcept` и операцию `noexcept` , можно устанавливать `noexcept` , зависящий от того, являются ли вызываемые внутри функции `noexcept` или нет.

```
template <class T>
auto Sum(const T& x, const T& y) noexcept(noexcept(x + y)) {
  return x + y;
}
```

В данном примере, для `T=int` `Sum` будет `noexcept` , а, например, для `T=std::string` не будет (выделение памяти для строки может завершиться неудачно).

31.6 Небросающий new

Использование `new` небезопасно с точки зрения возможных исключений - если недостаточно памяти, то вылетает исключение типа `std::out_of_memory` .

```
// potentially throw an exception
auto ptr = new int[1'000'000];
```

Это поведение можно изменить - можно попросить вместо исключения возвращать `nullptr` с помощью следующего синтаксиса:

```
auto ptr = new(std::nothrow) int[1'000'000];
if (!ptr) { /* not enough memory */ }
```

Это бывает полезно, если не хочется возиться с обработкой исключений (писать `try-catch`)

31.7 Исключения в конструктарах

Рассмотрим следующий код:

```
struct A {
  std::vector<int> v;
  int* ptr;

  A() : v(100), ptr(new int) {
    f(); // potentially throws an exception
  }

  ~A() {
    delete ptr;
  }
};
```

Так как конструктор не завершил работу, объект не считается созданным, а это значит, что деструктор вызван не будет! Утечка памяти (ptr)

Решение: перехватить исключение, очистить память и бросить исключение дальше.

```
struct A {
  std::vector<int> v;
  int* ptr;

  A() : v(100), ptr(new int) {
    try {
      f(); // potentially throws an exception
    } catch (...) {
      delete ptr;
      throw;
    }
  }

  ~A() {
    delete ptr;
  }
};
```

С вектором все будет нормально - объект вектора был создан, а значит для него будет вызван деструктор.

Если вы используете RAII, то проблем совсем нет.

```
struct A {
  std::vector<int> v;
  std::unique_ptr<int> ptr;

  A() : v(100), ptr(std::make_unique<int>(0)) {
    f(); // potentially throws an exception
  }
};
```

31.8 Исключение в деструкторах

Очевидная проблема - утечка ресурсов, которую решить довольно просто:

```
struct A {
  std::vector<int> v;
  int* ptr;

  // ...

  ~A() {
    try {
      f(ptr); // potentially throws an exception
    } catch (...) {
      delete ptr;
      throw;
    }
    delete ptr;
  }
};
```

При этом для вектора в любом случае будет вызван деструктор.
Однако могут возникнуть проблемы, например, в следующем коде:

```
void h() {
  A a; // The destructor can throw an exception
  g(); // UB, if an exception is thrown
  // ...
}
```

Сценарий такой: допустим g() бросает исключение, начинается раскручивание стека, вызывается деструктор A и в нем снова бросается исключение. Итог - из одной функции летит 2 исключения (UB).

Таким образом, не позволяйте исключениям покидать деструкторы.

В современном C++ все деструкторы по умолчанию помечены как noexcept, поэтому вылет исключения из деструктора приводит к аварийному завершению программы.

31.9 Гарантии безопасности исключений

Давайте формализуем понятие "безопасный код"с помощью так называемых гарантий безопасности.

Гарантия безопасности отвечает на вопрос, в каком состоянии находится система после возникновения ошибки.

Гарантии безопасности:

1. Гарантия отсутствия исключений
2. Базовая гарантия безопасности
3. Строгая гарантия безопасности

31.9.1 Гарантия отсутствия исключений

Самая простая (для понимания) гарантия. Функция удовлетворяет гарантии отсутствия исключений, если она никогда не бросает исключений.

Примеры:

```
template <class T>
size_t Stack<T>::Size() const noexcept { // satisfies the guarantee
  return size_;
}

template <class T>
Stack<T>::Stack(const Stack<T>&); // does not satisfy guarantee
// can quit, for example, when there is not enough memory
```

31.9.2 Базовая гарантия безопасности

Функция удовлетворяет базовой гарантии безопасности, если после возникновения исключения все компоненты программы находятся в согласованном (валидном состоянии), утечки ресурсов не произошло.

31.9.3 Строгая гарантия безопасности

Функция удовлетворяет строгой гарантии безопасности, если после возникновения исключения все компоненты программы находятся **в том же состоянии**, что и до вызова, утечки ресурсов не произошло.

31.9.4 Применение noexcept

К сожалению, "эффективный код"и "безопасный код"далеко не всегда совместимы друг с другом.

Часто для достижения безопасности приходится обходиться без наиболее эффективных функций (в них могут возникать неожиданные ошибки).

noexcept позволяет дать понять другим функциям, что она безопасна, и что ее можно спокойно использовать, не переживая за возможные ошибки.

31.10 Иерархия исключений C++

31.10.1 Исключения-классы

В C++ есть договоренность - бросать исключения классовых типов (тип дает информацию о природе исключений)

Давайте сделаем следующее: заведем общий базовый класс Exception , а все остальные исключения будем наследовать от него.

```
class Exception {
 public:
  virtual const char* What() const noexcept { return "Exception"; }
  virtual ~Exception() = default;
};

class DivisionByZero : public Exception {
  // ...
  const char* What() const noexcept override { return "DivisionByZero";
};

int main() {
  try {
    Divide(1, 0);
  } catch (const Exception& ex) {
    std::cerr << ex.What() << '\n'; // DivisionByZero
  }
}
```

31.10.2 Стандартная библиотека исключений C++

В C++ есть готовый базовый класс исключений - `std::exception` .

Он содержит единственный виртуальный метод - `what()` .

Все стандартные классы исключений унаследованы от него: `std::logic_error` , `std::runtime_error` , `std::bad_cast` (бросает `dynamic_cast`), `std::bad_alloc` (бросает `new` при неудаче), `std::bad_weak_ptr` и много других

От них, в свою очередь, могут быть унаследованы (уточнены) другие исключения.

Например, `std::out_of_range` унаследован от `std::logic_error` , а `std::bad_any_cast` от `std:bad_cast`

Таким образом, можем группировать исключения по степени общности и по смыслу.

Свои классы исключений также стоит наследовать от одного из стандартных классов ошибок

```
class DivisionByZero : public std::runtime_error {
 public:
  using std::runtime_error::runtime_error;
};
```

32 Итераторы

Итератор - объект с интерфейсом указателя, предоставляющий доступ к элементам контейнера и возможность их обхода.

32.1 Неконстантные итераторы

Все контейнеры поддерживают методы `begin` (итератор на начало контейнера) и `end` (итератор на элемент следующий за последним).

Они имеют тип `std::vector<int>::iterator`, `std::list<std::string>::iterator` и т.п.

Чтобы абстрагироваться от понятия контейнера, большинство алгоритмов принимают непосредственно итераторы (полуоткрытый интервал) для работы с последовательностями.

```
template <class Iterator>
void Print(Iterator begin, Iterator end) {
  for (; begin != end; ++begin) {
    std::cout << *begin << '\n';
  }
}

std::vector<int> v{3, 4, 2, 1};

std::sort(v.begin(), v.end());
Print(v.begin(), v.end()); // [1, 2, 3, 4]
```

С помощью итераторов можно изменять элементы, на которые они указывают (иногда).

```
template <class Iterator>
void ZeroAll(Iterator begin, Iterator end) {
  for (; begin != end; ++begin) {
    *begin = 0;
  }
}

std::vector<int> v{1, 2, 3};
ZeroAll(v.begin(), v.end()); // [0, 0 ,0]
```

32.2 Константные итераторы

Константные итераторы позволяют читать значения, на которые указывают, но не позволяют изменять (при разыменовании возвращают константную ссылку).

```
template <class Iterator>
void ZeroAll(Iterator begin, Iterator end) {
  for (; begin != end; ++begin) {
    *begin = 0; // CE: assignment of read-only location
  }
}

const std::vector<int> cv{1, 2, 3};
ZeroAll(cv.begin(), cv.end());
```

`begin()` и `end()` теперь возвращают `std::vector<T>::const_iterator`.

Для получения константных итераторов у неконстантных контейнеров можно использовать методы `cbegin()` и `cend()` .

32.3 Отличия итераторов

В чем отличие `std::vector<T>::const_iterator` и `const std::vector<T>::iterator` ?

В том же, в чем отличие `const T*` и `T* const` :

```
std::vector<int>::iterator it = v.begin();
++it; *it = 0;

std::vector<int>::const_iterator it = v.cbegin();
++it; // *it = 0; CE

const std::vector<int>::iterator it = v.begin();
*it = 0; // ++it; CE

const std::vector<int>::const_iterator it = v.cbegin();
// ++it; *it = 0; CE
```

Существует преобразование из `iterator` в `const_iterator` , но не наоборот.

32.4 Категории итераторов

Любой итератор обязан определять операции `++` , унарный `*` , `->` .

В зависимости от дополнительных поддерживаемых операций итераторы могут принадлежать следующим категориям:

- input iterator (итератор ввода)
- output iterator (итератор вывода)
- forward iterator (прямой итератор)
- bidirectional iterator (двунаправленный итератор)
- random access iterator (итератор произвольного доступа)
- contiguous iterator (непрерывный итератор) (C++20)

32.4.1 Input Iterator

Input iterator (итератор ввода) - итератор, предоставляющий доступ на чтение элементов (с помощью разыменования `*` или операции `->`).

Объекты итератора ввода могут быть проверены на равенство (`==` , `!=`).

Данные итераторы являются однопроходными, то есть можно пройти в одном направлении ровно 1 раз.

```cpp
template <class InputIterator>
void Function(InputIterator begin) {
  auto copy_begin = begin;
  ++begin; // Ok, copy_begin is invalidated
  ++copy_begin; // UB
}
```

Примеры: `std::istream_iterator`

32.4.2 Forward Iterator

Forward Iterator (прямой итератор) - input iterator, с возможностью создания по умолчанию (нулевой итератор) и многократным проходом по последовательности.

```cpp
template <class ForwardIterator>
void Function(ForwardIterator begin, ForwardIterator end) {
  auto copy_begin = begin;
  for (; begin != end; ++begin) {
    std::cout << *begin << '␣';
  }
  for (; copy_begin != end; ++copy_begin) {
    std::cout << *copy_begin << '␣';
  }
}
```

Примеры: `std::forward_list<T>::iterator` , `std::unordered_set<T>::iterator` , `std::unordered_multimap<Key, Value>::iterator` , ...

32.4.3 Bidirectional Iterator

Bidirectional iterator (двунаправленный итератор) - forward iterator, для которого дополнительно определены операции `--` (префиксная и постфиксная).

```cpp
template <class BidirectionalIterator>
void Function(BidirectionalIterator begin, BidirectionalIterator end) {
  for (auto it = begin; it != end; ++it) {
    std::cout << *it << '␣';
  }
  for (auto it = end; it != begin;) {
    std::cout << *(--it) << '␣';
  }
}
```

Примеры: `std::list<T>::iterator` , `std::set<T>::iterator` , `std::multimap<Key, Value>::iterator` , ...

32.4.4 Random Access Iterator

Random access iterator (итератор произвольного доступа) - bidirectional iterator, для которого дополнительно определены арифметические операции (сложение с числом `+` , `+=`

, вычитание числа `-` , `-=` , разность итераторов `-`), доступ по индексу `[]` , а также отношение порядка (`<` , `>` , `<=` , `>=`)

```
template <class RndAccessIterator>
void Function(RndAccessIterator begin, RndAccessIterator end) {
  for (auto it = begin; it < end; it += 2) {
    std::cout << *it << '␣';
  }
  size_t size = end - begin;
  for (size_t i = 0; i < size; ++i) {
    std::cout << begin[i] << '␣';
  }
}
```

Примеры: `std::vector<T>::iterator` , `std::array<T, N>::iterator` , `std::deque<T>::iterator`

32.4.5 Contiguous Iterator (C++20)

Contiguous iterator (непрерывный итератор) - random access iterator, для которого выполнено свойство:

```
*(iterator + n) <=> *(&(*iterator) + n)
```

То есть данные под итератором расположены непрерывно в памяти.
Примеры: `std::vector<T>::iterator` , `std::array<T, N>::iterator`

32.4.6 Output iterator

Любой из указанных выше итераторов дополнительно может принадлежать категории **output iterator (итератор вывода)**, если результату разыменования можно присвоить значение.

```
OutputIterator iterator = ...;
*iterator = 0;
```

32.5 Обобщенные функции для работы с итераторами

При работе с итераторами используются следующие обобщенные функции:

- `std::advance(it, n)` - продвигает it на n шагов вперед
- `std::next(it)` - возвращает итератор на следующий элемент
- `std::next(it, n)` - // возвращает итератор на n шагов вперед
- `std::prev` - аналогичен `std::next` , но шагает назад
- `std::distance(begin, end)` - расстояние между begin и end

32.6 Инвалидация итераторов

Итератор, указывающий на недействительные данные или потенциально являющийся таковым, является невалидным. Такой итератор не может быть разыменован и использован (приводит к undefined behaviour).

Классический пример - расширение буфера в `std::vector` :

```
std::vector<int> v{1, 2, 3};
auto iterator = v.begin() + 1;
for (int i = 0; i < 100; ++i) {
// surely there will be a redistribution
  v.push_back(i);
}
*iterator = 0; // undefined behaviour
```

32.7 Range-based for

В C++11 появился следующий способа обхода контейнеров:

```
std::vector<int> v{1, 2, 3, 4};
for (int x : v) {
  std::cout << x << ' ';
}

std::vector<int> v{1, 2, 3, 4};
for (int& x : v) {
  x = 0;
}

std::map<int, std::string> m{{1, "one"}, {2, "two"}, {3, "three"}};
for (const auto& item : m) {
  std::cout << item->first << ": " << item->second << '\n';
}
```

32.7.1 Принцип работы

Цикл вида

```
for (<type> value : container) {
  // ...
}
```

Эквивалентен следующему коду (он подставляется неявно компилятором):

```
for (auto it = container.begin(), end = container.end(); it != end; ++i
  <type> value = *it;
  // ...
}
```

32.7.2 Range-based for для C-массивов

У C-массивов нет методов `.begin()` и `.end()` . Но можно реализовать внешние функции `std::begin(container)` и `std::end(container)` , которые вызывают `begin` и `end` , если они есть. Если их нет, то работает специализация определенная для C-массивов и узнает размер массива через `sizeof` .

```
for (auto it = std::begin(container), end = std::end(container);
  it != end; ++it) {
    <type> value = *it;
    // ...
}

template <class T, size_t N>
T* begin(T (&array)[N]) {
  return array;
}

template <class T, size_t N>
T* end(T (&array)[N]) {
  return array + N;
}
```

32.7.3 Range-based for: собственные контейнеры

Чтобы range-based for работал с пользовательскими контейнерами необходимо:

- Либо реализовать методы `.begin()` , `.end()` , возвращающие итераторы
- Либо реализовать внешние функции `begin(container)` , `end(container)`

В зависимости от того, что будет найдено компилятором, будет использован первый или второй вариант.

33 Вычисление на этапе компиляции

Известно, что в качестве размера C-style массива или параметра шаблона могут выступать только константы, значения которых известны на этапе компиляции (константные выражения).

Константное выражение (constant expression) - выражение, которое может быть вычислено на этапе компиляции.

Например, в качестве размера C-style массива или параметра шаблона могут выступать только константные выражения.

```cpp
int main() {
  const int n = 100;
  int c_style[n];  // Ok
  std::array<int, n * n + 32> arr;  // Ok
  return 0;
}
```

33.1 Проблема

А что если хочется передать не константу, а значение функции от константы?

```cpp
int64_t Pow(int64_t x, int n);  // x^n

int main() {
  const int n = 10;
  int c_style[Pow(3, n)];  // CE
  std::array<int, Pow(5, n)> arr;  // CE
  return 0;
}
```

Ошибки:

```
main.cpp: error: ISO C++ forbids variable length array c_style
[-Werror=vla] int c_style[Pow(3, n)];  // CE
main.cpp: error: call to non-constexpr function
int Pow(int, int) std::array<int, Pow(5, n)> arr;  // CE
```

Таким образом, необходимо заставить компилятор вычислять самостоятельно?

33.2 Классические compile-time вычисления (C++03)

33.2.1 Подготовка

Заведем специальный шаблонный класс, который хранит в себе определенную целочисленную константу:

```cpp
template <class Integer, Integer Value>
struct IntegralConstant {
  static const Integer kValue = Value;
  // ...
};

int main() {
  int c_style[IntegralConstant<int, 100>::kValue];
  std::array<int, IntegralConstant<int, 100>::kValue> arr;
}
```

А лучше воспользуемся готовым из стандартной библиотеки

```cpp
#include <type_traits>  // std::integral_constant

int main() {
  int c_style[std::integral_constant<int, 100>::value];
  std::array<int, std::integral_constant<int, 100>::value> arr;
}
```

33.2.2 Идея решения

Что гарантированно происходит на этапе компиляции?

Правильно! - Подстановка шаблонных параметров.

То есть компилятор обязан вычислить значение шаблонного параметра, иначе у него не получится инстанцировать шаблонный класс.

```cpp
template <int N>
struct Square : std::integral_constant<int, N * N> {};

int main() {
  int c_style[Square<100>::value];  // Ok
  std::array<int, Square<100>::value> arr; // Ok
}
```

33.2.3 Примеры

Достаточно простая реализация Факториала:

```cpp
//Factorial
template <size_t N>
struct Factorial
  : std::integral_constant<size_t, N * Factorial<N - 1>::value> {};

// Problem 1: missing end of recursion.
template <>
struct Factorial<0> : std::integral_constant<size_t, 1> {};

// Problem 2: Factorial<N>::value is cumbersome.
template <size_t N>
struct Factorial;  // forward declaration

template <size_t N>
inline const size_t kFactorialV = Factorial<N>::value;
// template variable

template <size_t N>
struct Factorial : std::integral_constant<size_t, N *
kFactorialV<N - 1>> {};

template <>
struct Factorial<0> : std::integral_constant<size_t, 1> {};

int main() {
  int c_style[kFactorialV<8>];
  std::array<int, kFactorialV<10>> arr;
}
```

Числа Фибоначчи:

Вычисляется на этапе компиляции! Так как шаблон с одним и тем же набором параметров не инстанцируется дважды. Во время исполнения, так как просто подставляется готовое значение.

```
//Fibonacci
template <size_t N>
struct Fibonacci;  // forward declaration

template <size_t N>
inline const size_t kFibonacciV = Fibonacci<N>::value;

template <size_t N>
struct Fibonacci
  : std::integral_constant<size_t, kFibonacciV<N - 1> +
  kFibonacciV<N - 2>> {};

template <>
struct Fibonacci<0> : std::integral_constant<size_t, 0> {};

template <>
struct Fibonacci<1> : std::integral_constant<size_t, 1> {};
```

Быстрое возведение в степень:
Можно сделать быстрое возведение следующим образом:

```
template <int64_t X, size_t N>
struct Pow;

template <int64_t X, size_t N>
inline const int64_t kPowV = Pow<X, N>::value;

template <int64_t X, size_t N>
struct Pow
  : std::integral_constant<int64_t, N % 2 == 0 ? kPowV<X * X, N / 2> :
                                   X * kPowV<X * X, (N - 1) / 2>> {};

template <int64_t X>
struct Pow<X, 0> : std::integral_constant<int64_t, 1> {};

int main() {
  int c_style[kPowV<3, 10>];
  std::array<int, kPowV<5, 10>> arr;
}
```

33.3 Современные compile-time вычисления (C++11)

33.3.1 constexpr функции

constexpr функция - функция, которая может быть вычислена на этапе компиляции.

```
constexpr size_t Factorial(size_t n) {
  size_t res = 1;
  for (; n > 1; --n) {
    res *= n;
  }
  return res;
}

int main() {
  int c_style[Factorial(5)];  // Ok
  std::array<int, Factorial(3)> arr;  // Ok
}
```

Свойства constexpr функций из C++17:

- Автоматически являются inline функциями.
- Не могут быть виртуальными (до C++20).
- Не могут иметь try-catch блок (до C++20).
- Не могут содержать ассемблерные вставки (до C++20).
- Не могут использовать операции с динамической памятью (до C++20).
- Не могут содержать goto (до C++23).
- Могут определять только нестатические литеральные переменные (до C++23).
- При вычислении на этапе компиляции UB = CE, исключение = CE.

33.3.2 Примеры

В C++17:

```
constexpr size_t Fibonacci(size_t n) {
  if (n <= 1) {
    return n;
  }
  size_t prev = 0;
  size_t res = 1;
  for (; n > 1; --n) {
    res += prev;
    prev = res - prev;
  }
  return res;
}

constexpr int64_t Pow(int64_t x, size_t n) {
  if (n == 0) {
    return 1;
  }
  return (n % 2 == 0 ? 1 : x) * Pow(x * x, n / 2);
}
```

В C++20 можно и так:

```
constexpr size_t Fibonacci(size_t n) {
  auto fib = new size_t[n + 1];
  fib[0] = 0;
  fib[1] = 1;
  for (size_t i = 2; i <= n; ++i) {
    fib[i] = fib[i - 1] + fib[i - 2];
  }
  auto res = fib[n];
  delete[] fib;
  return res;
}

constexpr size_t Fibonacci(size_t n) {
  std::vector<size_t> v(n);
  v[0] = 0;
  v[1] = 1;
  // ...
}
```

33.3.3 Литеральные типы

В constexpr функции можно определять только переменные литерального типа.

Литеральный тип (неформально) - либо примитивный тип, либо тип с constexpr конструктором (кроме конструктора копирования и перемещения) и деструктором, либо агрегат из литеральных типов.

```cpp
int x = 0;  // Ok

struct S {
  int a;
  double b;
};

S y{};  // Ok

std::array<int, 5> z{}; // Ok

std::vector<int> t;  // CE (Ok C++20)
```

33.4 consteval (C++20)

Если constexpr функция вызывается в контексте, где ожидается константное выражение, то ее значение будет вычислено на этапе компиляции.

Но! В прочих ситуациях тот факт, что функция может быть вычислена на этапе компиляции, не гарантирует, что она будет вычислена во время компиляции.

```cpp
constexpr int64_t Pow(int64_t x, size_t n);

int main() {
  // Will be calculated at compile time
  int c_style[Pow(3, 5)];

  // Most likely to be called during execution
  std::cout << Pow(5, 3) << '\n';

  // Will be called during execution
  std::cout << Pow(std::rand(), 2) << '\n';
}
```

Чтобы гарантировать вычисление на этапе компиляции, можно воспользоваться consteval.

```cpp
consteval int64_t Pow(int64_t x, size_t n);

int main() {
  // Will be calculated at compile time
  int c_style[Pow(3, 5)];

  // Will be calculated at compile time
  std::cout << Pow(5, 3) << '\n';

  std::cout << Pow(std::rand(), 2) << '\n';  // CE
}
```

33.5 constexpr переменные

constexpr переменная - константная переменная, значение которой известно на этапе компиляции.

```
constexpr int64_t x = 3;
constexpr auto x5 = Pow(x, 5);

x = 5;  // CE

constexpr int y;  // CE
constexpr int z = std::rand();  // CE
```

Свойства constexpr переменных:

- переменная должна быть литерального типа
- она должна быть проинициализирована константным выражением
- являются константами

А что если хочется проинициализировать константой времени компиляции, но при этом не делать саму переменную константой?

```
constexpr int64_t Pow(int64_t x, size_t n);

// ...

// not guaranteed to calculate Pow at compile-time
auto x = Pow(3, 5);
```

33.6 constinit переменные (C++20)

constinit гарантирует, что начальное значение переменной будет вычислено на этапе компиляции. При этом сама переменная константой не считается.

```
constexpr int64_t Pow(int64_t x, size_t n);

// ...

// guaranteed to calculate Pow in compile-time
constinit auto x = Pow(3, 5);
x = 10;  // Ok
std::cin >> x;  // Ok
```

34 Классическое метапрограммирование

34.1 Частичная специализация шаблонов классов

Иногда хочется задать определенное поведение класса не для конкретного типа, а для целого семейства типов (например, для указателей).

```cpp
template <class T>  // general template
struct IsPointer {
  inline static const bool value = false;
  static bool IsIntPointer() { return false; }
};

template <class T>  // partial specialization
struct IsPointer<T*> {
  inline static const bool value = true;
  static bool IsIntPointer() { return false; }
};

template <>  // full specialization
struct IsPointer<int*> {
  inline static const bool value = true;
  static bool IsIntPointer() { return true; }
};
```

Более специализированная версия всегда побеждает менее специализированную

```cpp
std::cout << IsPointer<int>::value << '␣'                // 0
          << IsPointer<int>::IsIntPointer() << '\n';     // 0

std::cout << IsPointer<char*>::value << '␣'              // 1
          << IsPointer<char*>::IsIntPointer() << '\n';   // 0

std::cout << IsPointer<int*>::value << '␣'               // 1
          << IsPointer<int*>::IsIntPointer() << '\n';    // 1
```

```cpp
template <class T, class U>
struct S {  // 1
};

template <class T>
struct S<T, int> {  // 2
};

template <class T>
struct S<float, T> {  // 3
};

S<bool, bool> a;          // 1
S<bool, int> b;           // 2
S<const float, bool> c;   // 1
S<float, int> d;          // CE
```

34.2 Определители типов

С помощью частичной специализации можно определять простейшие свойства типов в шаблонном контексте:

```
template <class T>
struct IsConst {
  static constexpr bool value = false;
};

template <class T>
struct IsConst<const T> {
  static constexpr bool value = true;
};
```

"Упростим"предыдущий пример:

```
template <class T>
struct IsConst : std::integral_constant<bool, false> {};

template <class T>
struct IsConst<const T> : std::integral_constant<bool, true> {};

template <bool B>
using bool_constant = integral_constant<bool, B>;
using true_type = bool_constant<true>;
using false_type = bool_constant<false>;

template <class T>
struct IsConst : std::false_type {};

template <class T>
struct IsConst<const T> : std::true_type {};
```

```
template <class T>
struct IsPointer : std::false_type {};

template <class T>
struct IsPointer<T*> : std::true_type {};

template <class T>
struct IsPointer<T* const> : std::true_type {};

template <class T>
struct IsPointer<T* volatile> : std::true_type {};

template <class T>
struct IsPointer<T* const volatile> : std::true_type {};
```

```
template <class T>
struct IsLvalueReference : std::false_type {};

template <class T>
struct IsLvalueReference<T&> : std::true_type {};

template <class T>
struct IsRvalueReference : std::false_type {};

template <class T>
struct IsRvalueReference<T&&> : std::true_type {};

template <class T>
struct IsReference
  : std::bool_constant<IsLvalueReference<T>::value || IsRvalueReference
```

```
template <class T, class U>
struct IsSame : std::false_type {};

template <class T>
struct IsSame<T, T> : std::true_type {};

template <class T, class... Other>
struct AreSame : std::bool_constant<(IsSame<T, Other>::value && ...)> {
```

34.2.1 Примеры применения

Замечание: описанные выше определители типов (`std::is_const, std::is_reference,...`), а также соответствующие им шаблонные переменные (`std::is_const_v, std::is_reference_v, ...`) есть в заголовочном файле `<type_traits>` .

```
template <class... Ints>
auto Sum(Ints... values) {  // sum of ints only
  static_assert(std::is_same_v<int, Ints> && ...);
  return (values + ...);
}

template <class Storage>
void ProcessAndDestroy(Storage storage) {
  // ...
  if (std::is_pointer_v<Storage>) {
    delete[] &storage[0];
  }
}
```

34.2.2 Примеры применения: if constexpr

Рассмотрим следующий код:

```
template <class Storage>
void ProcessAndDestroy(Storage storage) {
  // ...
  if (std::is_pointer_v<Storage>) {
    delete[] storage;
  }
}

ProcessAndDestroy(std::vector<int>{1, 2, 3, 4, 5});
ProcessAndDestroy(new int[5]{1, 2, 3, 4, 5});
```

Блок внутри `if` компилируется в любом случае!

```
main.cpp: error: type 'std::vector<int>' argument given to 'delete',
expected pointer 10 |     delete[] storage;
```

if constexpr (C++17) - конструкция, позволяющая "отключать"ветки инстанцирования. Принимает булевское константное выражение.

```
template <class Storage>
void ProcessAndDestroy(Storage storage) {
  // ...
  if constexpr (std::is_pointer_v<Storage>) {
    delete[] storage;
  }
}

ProcessAndDestroy(std::vector<int>{1, 2, 3, 4, 5});  // Ok
ProcessAndDestroy(new int[5]{1, 2, 3, 4, 5});
```

```
template <class Iterator>
auto Get(Iterator iterator, size_t n) {
  using Tag = typename
    std::iterator_traits<Iterator>::iterator_category;
  if constexpr (std::is_base_of_v<std::random_access_iterator_tag, Tag>
    return iterator[n];
  } else {
    while (n--) {
      ++iterator;
    }
    return *iterator;
  }
}
```

34.3 Модификаторы типов

```
template <class T>
struct RemoveConst {
  using type = T;
};

template <class T>
struct RemoveConst<const T> {
  using type = T;
};

template <class T>
auto CreateCopy(T* array, size_t n) {
  auto copy = new RemoveConst<T>::type[n];
  std::copy(array, array + n, copy);
  return copy;
}
```

Упростим код:

```
template <class T>
struct TypeIdentity {  // std::type_identity
  using type = T;
};

template <class T>
struct RemoveConst : TypeIdentity<T> {};
template <class T>
struct RemoveConst<const T> : TypeIdentity<T> {};

template <class T>
using RemoveConstT = typename RemoveConst<T>::type;

template <class T>
auto CreateCopy(T* array, size_t n) {
  auto copy = new RemoveConstT<T>[n];
  std::copy(array, array + n, copy);
  return copy;
}
```

```cpp
template <class T>
struct RemoveReference : std::type_identity<T> {};

template <class T>
struct RemoveReference<T&> : std::type_identity<T> {};

template <class T>
struct RemoveReference<T&&> : std::type_identity<T> {};

template <class T>
using RemoveReferenceT = typename RemoveReference<T>::type;
```

34.4 std::move / std::forward

Наконец-то можем разобрать реализацию функций std::move и std::forward

```cpp
template <class T>
std::remove_reference_t<T>&& move(T&& value) {
  return static_cast<std::remove_reference_t<T>&&>(value);
}

template <class T>
T&& forward(std::remove_reference_t<T>& value) {
  return static_cast<T&&>(value);
}
```

Что делает std::move? - преобразует к rvalue ссылке (xvalue)
Что принимает std::move? - что угодно (lvalue/rvalue)
Что возвращает std::move? - rvalue ссылку (xvalue)
Что делает std::forward? - условный std::move
Что принимает std::forward? - lvalue ссылку
Что возвращает std::forward? - lvalue/rvalue ссылку в зависимости от T

34.5 decltype / declval

Спецификатор decltype позволяет узнать тип сущности по идентификатору, а также тип и категорию значения произвольного выражения.

```cpp
int x = 0;
const float y = 1;
const int& rx = x;

decltype(x) a = 0;   // int
decltype(y) b = 1;   // const float
decltype(rx) c = a;  // const int&
```

```
template <class T>
auto ExtractNested(const std::vector<T>& v) {
  std::vector<decltype(v[0].x)> res;
  res.reserve(v.size());
  for (const auto& value : v) {
    res.push_back(value.x);
  }
  return res;
}
```

Если подать decltype выражение (не являющееся именем переменной или поля), то он вернет:

- просто тип, если категория выражения - prvalue
- тип с & , если категория выражения - lvalue
- тип с && , если категория выражения - xvalue

```
int x = 0;

decltype(x + x);          // int
decltype(++x);            // int&
decltype(x++);            // int
decltype((x));            // int&
decltype(std::move(x));   // int&&
```

auto выводит тип в соответствии с правилами вывода шаблонных параметров:

```
volatile int x = 0;
auto y = x;     // int
auto z = ++y;   // int
auto t = (x);   // int

But you can force the type to be inferred according
to the decltype rules:

volatile int x = 0;
decltype(auto) y = x;     // volatile int
decltype(auto) z = ++y;   // volatile int&
decltype(auto) t = (x);   // volatile int&
```

```
template <class T>
decltype(auto) move(T&& value) {
  return static_cast<std::remove_reference_t<T>&&>(value);
}

template <class T>
decltype(auto) forward(std::remove_reference_t<T>& value) {
  return static_cast<T&&>(value);
}
```

```
template <class T, class U, class N>
auto Sum(const std::array<T, N>& lhs, const std::array<U, N>& rhs) {
  std::array<std::remove_cvref_t<decltype(lhs[0] + rhs[0])>, N> res;
  for (size_t i = 0; i < N; ++i) {
    res[i] = lhs[i] + rhs[i];
  }
  return res;
}
```

Добавим noexcept в зависимости от того, может ли сумма бросить исключение или нет:

```
template <class T, class U, class N>
auto Sum(const std::array<T, N>& lhs, const std::array<U, N>& rhs)
noexcept(noexcept(lhs[0] + rhs[0])) {

  std::array<std::remove_cvref_t<decltype(lhs[0] + rhs[0])>, N> res;
  for (size_t i = 0; i < N; ++i) {
    res[i] = lhs[i] + rhs[i];
  }
  return res;
}
```

Допустим, хотим создать переменную, которая имела бы тот же самый тип, что и сложение двух объектов типа T:

```
T x = ...;
T y = ...;
decltype(x + y) z;  // had to create x and y for this

// how do we know if it can be called by default
decltype(T() + T()) z;
```

Функция std::declval позволяет "во что бы то ни стало"обратиться к объекту данного типа в невычисляемых контекстах (unevaluated context).

Забавный факт в том, что у нее нет определения! (Оно и не нужно)

```
decltype(std::declval<T>() + std::declval<T>()) z;
```

Пояснение: declval говорит "да-да я возвращаю то, что нужно". Компилятор не проверяет истинность этого, так как реально эта функция никогда не вызывается!

```
template <class T>
T&& declval();
```

35 Зависимые и независимые имена

Зависимое имя - имя, которое зависит от параметра шаблона.

Независимое имя - имя, которое не является зависимым.

```
template <class T>
struct A {
  static double x;
};

struct B {
  static int y;
};

template <class T>
void f() {
  B::y;         // independent name
  A<long>::x;   // independent name
  A<T>::x;      // dependent name
}
```

35.1 Зависимые имена типов

```
template <class T>
struct A {
  using X = T;
};

template <>
struct A<int> {
  const static int X = 0;
};

template <class T>
void f(T y) {
  A<T>::X* z;  // The compiler considers A<T>::X as a static field
}
```

error: dependent-name 'A<T>::X' is parsed as a non-type, but instantiation yields a type
Благо в следующей строчке он подсказывает, что нужно сделать:
note: say 'typename A<T>::X' if a type is meant

35.1.1 typename для зависимых имен типов

Чтобы зависимое имя воспринималось компилятором как тип, об этом нужно явно попросить:

```
template <class T>
struct A {
  using X = T;
};

template <>
struct A<int> {
  const static int X = 0;
};

template <class T>
void f(T y) {
  typename A<T>::X* z;  // typename indicates that A<T>::X is a type
}
```

typename не ставится, если по контексту понятно, что это тип, а не член

```
template <class T>
struct B : A<T>::X { ... }; // inheritance is only possible from a type

A<T>::X::type; // :: operator is only used with types

template <class T>
class S {
  using X = int;

  void f() {
    S<T>::X* y; // current instantiation, no ambiguity
  }
}
```

35.2 Зависимые имена шаблонов

Рассмотрим следующий код:

```
template <class T>
struct S {
  template <class U>
  void f(int) {}
};

template <>
struct S<int> {
  int f = 0;
};

template <class T>
void g() {
  S<T> s;
  s.f<int>(0); // CE
}
```

error: expected primary-expression before 'int'

Дело в том, что компилятор не знает, что представляет из себя S<T> (какой тип T будет подставлен). Он предполагает по умолчанию, что s.f - это поле класса, а < - это оператор "меньше".

35.2.1 template для зависимых имен шаблонов

Как и ранее, нужно дать подсказку компилятору. В данном случае - дать указание, что имя является именем шаблона:

```
template <class T>
void g() {
  S<T> s;
  s.template f<int>(0); // now the compiler knows that f is a template
}
```

35.3 Зависимые базовые классы

```
template <class T>
struct Base {
  int x;
};

template <class T>
struct Derived : Base<T> {
  void f() {
    x = 0;
  }
};
```

error: 'x' was not declared in this scope

Base<T> - неизвестный тип, так как тип T не определен. Поэтому компилятор не знает, что в Derived есть поле x.

35.3.1 this для зависимых базовых классов

Снова натолкнем компилятор на истинный путь:

```
template <class T>
struct Base {
  int x;
};

template <class T>
struct Derived : Base<T> {
  void f() {
    Base<T>::x = 0; // for example, this way
    this->x = 0;    // or this way
  }
};
```

36 SFINAE

36.1 SFINAE

SFINAE (Substitution Failure Is Not An Error) - "неудачная подстановка - не ошибка"правило языка, которое гласит, что если в результате подстановки шаблонных параметров в объявление функции или частичной специализации класса возникает некорректная конструкция, то это не приводит к ошибке компиляции. Компилятор просто отбрасывает этот шаблон/специализацию из списка кандидатов.

```
template <class T>
T& F(int);

template <class T>
void F(long);

// char& F(int) (wins because the argument type is a better match)
F<char>(0);

// void F(long) (the first one is not considered - void&
does not exist)
F<void>(0);
```

```cpp
template <class T>
struct IsReferenceable : std::false_type {};

template <class T>
struct IsReferenceable<T&> : std::true_type {};

IsReferenceable<int>::value;   // true

// false (specialization is not considered)
IsReferenceable<void>::value;
```

36.1.1 SFINAE: еще примеры

```cpp
template <size_t N, class = int[N % 2]> void A(int);  // 1
template <size_t N> void A(long);  // 2

A<5>(0);  // 1
A<2>(0);  // 2 (C-style arrays of size zero are not allowed)

template <class T> typename T::value_type B(int);  // 1
template <class T> T B(long);  // 2

B<std::vector<int>>(0);  // 1
B<int>(0);  // 2

template <class T> auto C(T x, T y) -> decltype(x + y);  // 1
template <class T, class U> T C(T x, U y);  // 2

C("aba", "caba");  // 2 (cannot add two const char*)
C(1, 2);  // 1
```

36.1.2 не SFINAE

Тело шаблонной функции не является SFINAE-контекстом!

```
template <size_t N> void A(int) { int arr[N]; }        // 1
template <size_t N> void A(long) { int arr[N + 1]; }  // 2

A<0>(0);  // CE (hard error)
A<5>(0);  // 1

template <class T> void B(int) { typename T::value_type x = 0; }
// 1
template <class T> void B(long) {} // 2

B<std::vector<int>>(0);  // 1
B<int>(0);  // CE (hard error)

template <class T> void C(T x, T y) { x + y; } ;  // 1
template <class T, class U> void C(T x, U y) {};  // 2

C("aba", "caba");  // CE (hard error)
C(1, 2);  // 1
```

36.1.3 SFINAE: применение не по назначению

SFINAE - полезный механизм языка, у которого был обнаружен интересный побочный эффект. С помощью него можно метапрограммировать!

```
template <class T>
struct IsReferenceable : std::false_type {};

template <class T>
struct IsReferenceable<T&> : std::true_type {};
```

36.2 Детекторы

С помощью SFINAE можно выявлять допустимость тех или иных конструкций с типами:

```
template <class T>
decltype(std::declval<T>().size(), std::true_type{}) Test(int)
noexcept;
template <class T>
std::false_type Test(long) noexcept;

template <class T>
struct HasSize : decltype(Test<T>(0)) {};

HasSize<int>::value;                // false
HasSize<std::vector<int>>::value;  // true
HasSize<int[5]>::value;             // false
```

36.2.1 IsAssignable

Задача: проверить, можно ли элементу одного типа присвоить элемент второго.

```
template <class T, class U>
decltype(std::declval<T>() = std::declval<U>(), std::true_type{})
Test(int) noexcept;

template <class, class>
std::false_type Test(long) noexcept;

template <class T, class U>
struct IsAssignable : decltype(Test<T, U>(0)) {};

IsAssignable<int, int>::value;     // false
IsAssignable<int&, int>::value;    // true
IsAssignable<int&, char>::value;   // true
IsAssignable<int&, char*>::value;  // false
```

36.2.2 IsNothrowMoveConstructible

Задача: проверить, существует ли небросающий конструктор перемещения.

```
template <class T>
std::bool_constant<noexcept(new(std::declval<T*>())
T(std::declval<T&&>()))>
Test(int) noexcept;

template <class, class...>
std::false_type Test(long) noexcept;

template <class T, class... Args>
struct IsNothrowMoveConstructible : decltype(Test<T>(0)) {};

IsNothrowMoveConstructible<int>::value;           // true
IsNothrowMoveConstructible<std::string>::value;   // true
IsNothrowMoveConstructible<std::istream>::value;  // false
```

36.2.3 IsPolymorphic

Задача: проверить, является ли класс полиморфным.

```
dynamic_cast to void* returns a pointer to the most derived class,
which is pointed to by the argument

template <class T>
decltype(dynamic_cast<const volatile void*>(std::declval<T*>()),
std::true_type{})
Test(int) noexcept;

template <class T>
std::false_type Test(long) noexcept;

template <class T>
struct IsPolymorphic : decltype(Test<T>(0)) {};
```

36.2.4 IsPublicBaseOf

Задача: проверить, является ли класс B публичным наследником A.

```
template <class T>
std::true_type ImaginaryFunction(T);

template <class A, class B>
decltype(ImaginaryFunction<A>(std::declval<B>())) Test(int) noexcept;

template <class A, class B>
std::false_type Test(long) noexcept;

template <class A, class B>
struct IsConvertible : decltype(Test<A, B>(0)) {};

template <class A, class B>
struct IsPublicBaseOf
  : std::bool_constant<std::is_class_v<A> && std::is_class_v<B> &&
                       IsConvertible<const volatile* A, B*>::value> {};
```

36.2.5 IsBaseOf*

Задача: проверить, является ли класс B наследником A.

```
template <class T>
std::true_type ImaginaryFunction(const volatile* T);
// there is inheritance

template <class>
std::false_type ImaginaryFunction(const volatile* void);
// no inheritance

template <class A, class B>
decltype(ImaginaryFunction<A>(std::declval<B*>())) Test(int) noexcept;

template <class A, class B>
// this is hit if inheritance is private
std::true_type Test(long) noexcept;

template <class A, class B>
struct IsBaseOf
  : std::bool_constant<std::is_class_v<A> && std::is_class_v<B> &&
                       decltype(Test<A, B>(0))::value> {};
```

std::enable_if

Часто хочется сделать так, чтобы в зависимости от каких-то условий компилятор игнорировал данный шаблон:

```
class A {
  A();
  A(const A&);
  A(A&&);

  template <class T>
  A(T&&);
};

A a;
auto copy = a;  // the template constructor with T=A& is selected
```

Можно прописать конструкторы на все ситуации (A&, const A&, A&&, const A&&). Но может проще просто "отключить"шаблон на эти ситуации?

std::enable_if - шаблонный класс, который параметризован булевским значением и типом T (по умолчанию - void), содержит внутренний тип type, если значение true и не содержит иначе.

```cpp
template <bool B, class T = void>
struct enable_if {
  using type = T;
};

template <class T>
struct enable_if<false, T> {};

template <bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

```cpp
template <class T>
std::enable_if_t<std::is_integral_v<T>, T> F(T x) { ... };  // 1

template <class T>
std::enable_if_t<std::is_floating_point_v<T>, T> F(T x) { ... };
// 2

F(0);  // 1
F(0.0);  // 2

template <class... Args, class =
std::enable_if_t<(std::is_arithmetic_v<Args> && ...)>>
auto Sum(Args... args) {
  return (args + ...);
}

Sum(0, 1, 4.5, 3);  // Ok
Sum(std::string("a"), std::string("ba"));  // CE
```

36.2.6 Решение заявленной проблемы

```cpp
class A {
  A();
  A(const A&);
  A(A&&);

  template <class T, class = std::enable_if_t<!std::is_same_v<A,
  std::remove_cvref_t<T>>>>
  A(T&&);
};

A a;
auto copy = a;  // A(const A&), the template is not considered
```

36.3 std::move_if_noexcept

Напомним, для чего используется данная функция (упрощенный пример):

```
template <class T>
void Realloc(T*& buffer, size_t size, size_t new_capacity) {
  const auto new_size = std::min(size, new_capacity);
  auto new_buffer = new T[new_capacity];
  try {
    for (size_t i = 0; i < new_size; ++i) {
      new_buffer[i] = std::move_if_noexcept(buffer[i]);
    }
  } catch (...) {
    delete[] new_buffer;
    throw;
  }
  delete[] std::exchange(buffer_, new_buffer);
}

If move is safe, then move; otherwise,
for strong exception safety, copy.
```

Наконец-то можем разобрать реализацию функции std::move_if_noexcept:

```
template <class T>
enable_if_t<is_nothrow_move_constructible_v<T> ||
!is_copy_constructible_v<T>, T&&>
move_if_noexcept(T& value) noexcept {
  return std::move(value);
}

template <class T>
enable_if_t<!is_nothrow_move_constructible_v<T> &&
is_copy_constructible_v<T>, const T&>
move_if_noexcept(T& value) noexcept {
  return value;
}

But it seems there is too much code duplication (in the return type).
```

36.4 std::conditional

std::conditional<B, T, F> - шаблонный класс, который параметризован булевским значением B и параметрами-типами T и F, определяет внутренний тип type = T или F в зависимости от истинности B.

```
template <bool B, class T, class F>
struct conditional {
  using type = F;
};

template <class T, class F>
struct conditional<true, T, F> {
  using type = T;
};

template <class B, class T, class F>
using conditional_t = typename conditional<B, T, F>::type;
```

36.5 Улучшение std::move_if_noexcept

Наконец-то можем разобрать реализацию функции std::move_if_noexcept:

```
template <class T>
conditional_t<is_nothrow_move_constructible_v<T> ||
!is_copy_constructible_v<T>,
T&&,
const T&>
move_if_noexcept(T& value) noexcept {
  return std::move(value);
}
```

36.6 std::void_t

Это класс, который превращает все (или почти все) в void.

```
template <class...>
using void_t = void;

void_t<>;  // void
void_t<int>;  // void
void_t<void, int, char&>;  // void

And why? How does this relate to SFINAE?
```

std::void_t можно использовать как тип, который существует, только при наличии каждого из его шаблонных параметров:

```
template <class T, class = void>
struct IsReferenceable : std::false_type {};

template <class T>
struct IsReferenceable<T, std::void_t<T&>> : std::true_type {};

template <class T, class = void>
struct HasSize : std::false_type {};

template <class T>
struct HasSize<T, std::void_t<decltype(std::declval<T>().size())>>
  : std::true_type {};
```

```
template <class T, class = void>
struct IsAssignable : std::false_type {};
template <class T>
struct IsAssignable<T, std::void_t<decltype(std::declval<T>()
= std::declval<T>())>> : std::true_type {};

template <class T, class = void>
struct HasValueType : std::false_type {};
template <class T>
struct HasValueType<T, std::void_t<typename T::value_type>> :
std::true_type {};

template <class T, class = void>
struct HasFirstSecond : std::false_type {};
template <class T>
struct HasFirstSecond<T,
                std::void_t<decltype(std::declval<T>().first),
                    decltype(std::declval<T>().second)>> {};

In short, std::void_t is a universal detector of
methods/fields/nested types.
```

37 Ограничения и концепты (C++20)

SFINAE позволяет проверить выражение на "корректность допустимы ли операции над данными операндами.

```
template <class T, class = void>
struct Addable : std::false_type {};

template <class T>
struct Addable<T,
  std::void_t<decltype(std::declval<T>() + std::declval<T>())>>
  : std::true_type {};
```

37.1 Проблема

Почему это выглядит так... не очень? Использование SFINAE - это костыль, эксплуатирование механизма языка под нужды, для которых он не был задуман.

```
template <class T, class = void>
struct Addable : std::false_type {};

template <class T>
struct Addable<T,
    std::void_t<decltype(std::declval<T>() + std::declval<T>())>>
  : std::true_type {};
```

37.2 requires выражение

requires выражение проверяет корректность вложенных выражений, а также выполнение требуемых свойств и возвращает true/false в зависимости от результата проверки.

`requires [(список параметров)] { <выражения/ограничения> }`

Выражения не вычисляются, а лишь анализируются!

```
template <class T>
inline constexpr bool is_addable = requires {
    std::declval<T>() + std::declval<T>(); };

template <class T>
inline constexpr bool is_subtractable = requires (T x) { x - x; };

std::cout << is_addable<int*> << ' ' << is_subtractable<int*> << '\n';

0 1
```

В списке параметров можно перечислять произвольный набор параметров (в том числе пакеты аргументов) и они не обязаны быть шаблонными.

```
template <class... Args>
inline constexpr bool is_addable = requires (Args... args, int x) {
  (args + ...) + x;  //
};
```

Список параметров не может содержать аргументы по умолчанию.

Если проверяемое условие ложно для любого шаблонного параметра (или вовсе не зависит от него), то программа считается некорректной (IFNDR).

```
template <class T>
inline constexpr auto is_addable = requires { "" + ""; };  // CE
```

requires выражение может содержать требования следующих видов:
1. Простые требования (simple requirements)
2. Типовые требования (type requirements)
3. Составные требования (compound requirements)
4. Вложенные требования (nested requirements)

37.2.1 Простые требования

Простое требование - произвольное выражение, которое требуется проверить на корректность.

Все требования в примере - простые (по одному на строку):

```
template <class T, class U>
inline constexpr bool example = requires (T x, U y) {
  x + y;
  x - y;
  x * y;
  x = y;
  F(x, y, x / y);
};
```

37.2.2 Типовые требования

Типовое требование - проверяет существование требуемого типа.

Начинается со слова typename, за которым следует имя типа:

```
template <class T, class U>
inline constexpr bool example = requires {
  typename std::vector<T>;
  typename T::value_type;
  typename A<T>::type;
};
```

37.2.3 Составные требования

Составное требование - позволяет проверить дополнительные свойства выражения.

`{ <выражение> } [noexcept] [ -> <ограничение> ]`

```
template <class T, class U>
inline constexpr bool example = requires (T x, U y) {
  { x + y; };  // same as a simple requirement
  { x * y; } noexcept;  // checks if multiplication is noexcept

  // checks if the result is convertible to int
  { x - y; } -> std::convertible_to<int>;

  // operation is noexcept and returns T
  { +x; } noexcept -> std::same_as<T>;
};
```

37.2.4 Вложенные требования

Вложенное требование - позволяет проверить выполнение некоторого условия (истина/-ложь).

`requires <условие>;`

```
template <class T, class U>
inline constexpr bool example = requires (T x, U y) {
  // checks if a variable can be instantiated
  std::is_base_of_v<T, U>;

  // checks the value of the variable
  requires std::is_base_of_v<T, U>;

  // checks the correctness of the record (always correct)
  sizeof(T) < 10;

  // checks the truth of the condition
  requires sizeof(T) < 10;

  // checks that T and U cannot be added
  requires !requires { x + y; };
};
```

37.3 Ограничения (C++20)

Ключевое слово requires может быть использовано в объявлении шаблона. В этом случае следующее за ним условие определяет множество типов, для которых этот шаблон объявлен.

```
template <class T>
requires (sizeof(T) <= 4) // can follow the template header
void Print(T) {
  std::cout << "Small type\n";
}
template <class T>
void Print(T x) requires (sizeof(x) > 4) {  // or follow the prototype
  std::cout << "Large type\n";
}

Print(0ll);  // Large type
```

Общий принцип похож на SFINAE: сначала отбираются кандидаты с пройденными требованиями, затем действуют правила выбора перегрузки.

```
struct A {
  A();
  A(const A&);
  A(A&&);

  template <class T>
  requires (!std::is_same_v<std::remove_cvref_t<T>, A>)
  A(T&&);
};

template <class Iterator> requires IsIterator<Iterator>
void Advance(Iterator& it, int n) {
  while (n-- > 0) ++it;
  while (n++ < 0) --it;
}

template <class Iterator> requires IsRandomAccessIterator<Iterator>
void Advance(Iterator& it, int n) {
  it += n;
}
```

```
template <size_t N>
requires (Fibonacci(N) > 1000)  // checks the condition
void F() { ... }  // 1

template <size_t N>
// checks if the expression is valid
requires requires { Fibonacci(N) > 1000; }
void F() { ... }  // 2
```

К сожалению, ограничения просто так не упорядочиваются компилятором по отношению "более частный-более общий":

```cpp
template <class T> requires (sizeof(T) <= 4)
void F(T);

template <class T> requires (sizeof(T) <= 4 && std::is_same_v<T, int>)
void F(T);

F(0);  // CE: ambiguous call, both templates match
```

Но вот, на чем можно ввести порядок, так это на концептах и их композициях.

37.4 Концепты (C++20)

Концепт - именованное требование.

```cpp
template <class T>
concept SmallType = sizeof(T) <= 4;

template <class T>
requires SmallType<T>
void F(T x);

// The last one can be rewritten as:

template <SmallType T>
void F(T x);

// and even as:

void F(SmallType auto x);
```

В качестве первого параметра концепта может выступать "подразумеваемый"параметр (очень полезно и удобно):

```cpp
template <class T, class U>
concept ConvertibleTo = std::is_convertible_v<T, U>;

template <ConvertibleTo<int> T>  // ConvertibleTo<T, int>
void F(T x);

template <ConvertibleTo<int>... Args>
int Sum(Args... args) {
  return (static_cast<int>(args) + ...);
}

requires {
  // requires ConvertibleTo<decltype(x + y), int>
  { x + y; } -> ConvertibleTo<int>;
}
```

Концепты можно объединять с помощью конъюнкций и дизъюнкций (&& и ||):

```
template <class T>
requires SmallType<T> && ConvertibleTo<T, int>
void F(T x);

// This can happen implicitly:

template <SmallType T> requires ConvertibleTo<T, int>
void F(T x, IsClass auto y) requires requires { x + y; };

// equivalently transforms into
template <class T, class U>
void F(T x, U y)
requires SmallType<T> && ConvertibleTo<T, int> &&
IsClass<U> && requires { x + y; };
```

Порядок на концептах

Каждое ограничение является либо атомарным ограничением (отдельным концептом либо булевским выражением), либо конъюнкцией ограничений, либо дизъюнкцией ограничений.

Два атомарных ограничений являются эквивалентными если они представляют собой одинаковые концепты (с одним набором фактических параметров), либо абсолютно одинаковые булевы выражения.

```
// these are different templates as the constraints are not equivalent!
template <class T> requires (sizeof(T) <= 4)
void F(T);

template <class T> requires (sizeof(T) < 5)
void F(T);
```

Ограничение A включает в себя (является более частным чем) ограничение B, если из A следует B (в смысле языка C++).

Формально:

A приводится к ДНФ, B - к КНФ

Атомарное ограничение A включает атомарное ограничение B если и только если A и B эквивалентны.

Дизъюнкт A включает в себя конъюнкт B если и только если в A есть атомарное ограничение, которое включает в себя некоторое атомарное ограничение из B.

A включает B если и только если каждый дизъюнкт из A включает в себя каждый конъюнкт из B.

37.4.1 Порядок на концептах: примеры

```
// atomic constraint
template <class T>
concept Iterator = requires (T x) { ++x; *x; };

// disjunctive constraint
template <class T>
concept BidirectionalIterator = Iterator<T> && requires (T x) { --
```

Примеры:

```
template <Iterator It>
void F(It);  // 1

template <BidirectionalIterator It>
void F(It);  // 2

std::vector<int> v{1, 2, 3, 4, 5};
std::forward_list<int> l{1, 2, 3, 4, 5};
F(v.begin());  // 2
F(l.begin());  // 1
```

```
// atomic constraint
template <class T>
concept Iterator = requires (T x) { ++x; *x; };

// disjunctive constraint
template <class T>
concept BidirectionalIterator = Iterator<T> && requires (T x) { --x; };

template <Iterator It>
void F(It);  // 1

template <BidirectionalIterator It>
void F(It);  // 2

std::vector<int> v{1, 2, 3, 4, 5};
std::forward_list<int> l{1, 2, 3, 4, 5};
F(v.begin());  // 2
F(l.begin());  // 1
```

```
// atomic constraint
template <class T>
concept Iterator = requires (T x) { ++x; *x; };

// atomic constraint
template <class T>
concept BidirectionalIterator = requires (T x) { ++x; *x; --x; };

template <Iterator It>
void F(It);  // 1

template <BidirectionalIterator It>
void F(It);  // 2

std::vector<int> v{1, 2, 3, 4, 5};
std::forward_list<int> l{1, 2, 3, 4, 5};
F(v.begin());  // CE: both match, no concept ordering
F(l.begin());  // 1
```

38 Ranges (C++20)

38.1 Проблема

Вспомним устройство стандартных алгоритмов C++:

```
template <class SomeIterator, class... SomeOtherArgs>
auto SomeAlgorithm(SomeIterator first, SomeIterator last,
    SomeOtherArgs...);

SomeAlgorithm(x.begin(), x.end(), ...);
```

где x - некоторый контейнер (или что угодно, у чего есть begin / end).

С одной стороны, это дает гибкость - можно применить алгоритм к произвольному промежутку контейнера.

С другой стороны, много лишнего кода для цельного контейнера.

38.2 Ranges (C++20)

Введем понятие диапазона (range). С точки зрения C++ диапазон - произвольный тип, у объектов которого можно вызвать begin и end:

```
template <class T>
concept range = requires (T& r) {
  std::ranges::begin(r);  // calls r.begin()
  std::ranges::end(r);    // calls r.end()
}
```

Аналогично итераторам диапазоны делятся по категориям (входной, выходной, прямой, двунаправленный, произвольного доступа, непрерывный):

```
template <class T>
using iterator_t = decltype(std::ranges::begin(std::declval<T&>()));

template <class T>
// range is an input range and has a forward iterator
concept forward_range =
  ranges::input_range<T> && std::forward_iterator<iterator_t<T>>;
```

38.3 Constrained algorithms (C++20)

В C++20 появились новые версии алгоритмов стандартной библиотеки, которые принимают диапазоны вместо пары итераторов:

```
template <class T, std::ranges::output_range R>
auto fill(R&& range, const T& value);

template <std::ranges::random_access_range R, class Cmp = std::less<>,
          class Proj = std::identity>
auto sort(R& range, Cmp cmp = {}, Proj proj = {});
```

```
std::vector<int> v(100);

std::ranges::fill(v, -1);
std::ranges::sort(v);
```

Сразу же бросается в глаза 2 ключевых изменения:
1. Параметры теперь явно ограничены концептами (constrained)
2. У sort (и многих других алгоритмов) появился третий аргумент - проектор.

Проектор необходим для того, чтобы можно было указывать ключ, по которому ведется сортировка, без переписывания компаратора (аналог аргумента key функции sorted в Python)

```
std::vector<std::string> v{...};

std::ranges::sort(v, {}, &std::string::size);  // pointer to method
std::ranges::sort(v, {}, [](const auto& s) { return s[0]; });
```

38.4 iterator_t и sentinel_t (C++20)

Посмотрим на код простейшего алгоритма count (упрощенный):

```
template <class InputIterator, class T>
size_t count(InputIterator first, InputIterator last, const T& value) {
  size_t res = 0;
  while (first != last) {
    if (*first++ == value) ++res;
  }
  return res;
}
```

Есть 2 проблемы:
1. Семантическая (на что-то наложено слишком строгое ограничение)
2. Функциональная (достаточно ли гибок этот алгоритм)

38.4.1 Проблема 1

```
template <class InputIterator, class T>
size_t count(InputIterator first, InputIterator last, const T& value) {
  size_t res = 0;
  while (first != last) {
    if (*first++ == value) ++res;
  }
  return res;
}
```

Данный алгоритм ожидает, что last будет являться входным итератором. Более того, он должен иметь тот же тип, что и first.

На самом деле, от last требуется лишь быть сравнимым на равенство с first!

Проблема 2

Задача: посчитать количество единиц в диапазоне от начала, до первого четного числа.

```
std::count(r.begin(),
    std::find_if(r.begin(), r.end(), [](auto x) {
        return x % 2 == 0; }), 1);
```

1. 2 раза проходим по одной последовательности (если бы писали цикл вручную, смогли бы все сделать за 1 проход)

2. А кто сказал, что по последовательности можно пройти 2 раза? (например, по итератору потокового ввода можно пройтись 1 раз!)

Напомним, что find_if и count работают с input итераторами. Но чтобы использовать их вместе требуется уже как минимум forward итератор.

38.4.2 Идея

Давайте позволим last иметь тип отличный от first.

Тогда в объект last можно будет заложить любое условие окончания итерирования!

```
template <class InputIterator, class Sentinel, class T>
size_t count(InputIterator first, Sentinel last, const T& value) {
  size_t res = 0;
  while (first != last) {
    if (*first++ == value) ++res;
  }
  return res;
}
```

```
struct CountedSentinel {
  size_t count;
};
struct TimedSentinel {
  clock_t start;
  clock_t limit;
};
struct EvenSentinel {};

template <class Iterator>
bool operator!=(Iterator, CountedSentinel s) {
  return --s.count != 0;
}

template <class Iterator>
bool operator!=(Iterator, TimedSentinel s) {
  return std::clock() - s.start < s.limit;
}

template <class Iterator>
bool operator!=(Iterator it, EvenSentinel) { return *it % 2 == 1; }

// count number of 1s in the first 10 elements
count(r.begin(), CountedSentinel{10}, 1);

// count for a limited time
count(r.begin(), TimedSentinel(std::clock(), 1e6), 1);

// count number of 1s until the first even number
count(r.begin(), EvenSentinel{}, 1);
```

38.4.3 sentine_t

В C++20 реализована описанная выше схема и в любой алгоритм можно передать пару итератор-ограничитель.

```
template <class R>
using iterator_t = decltype(std::ranges::begin(std::declval<T&>()));

template <class R>
using sentinel_t = decltype(std::ranges::end(std::declval<T&>()));

template <std::random_access_iterator I, std::sentinel_for<I> S,
          class Cmp = std::less<>, class Proj = std::identity>
auto sort(I first, S last, Cmp cmp = {}, Proj proj = {});
```

```
std::sentinel_for<S, I>                    ,
                        S      I .
```

38.5 borrowed_range / borrowed_iterator (C++20)

Мы разрешили передавать в алгоритмы диапазоны как объекты.

К чему это приводит?

```
// passing a temporary object, returns iterator to the temporary object
auto x = std::ranges::find(std::vector{1, 2, 3, 4, 5}, 3);
// passing a temporary object, returns iterator to the global scope
auto y = std::ranges::find(std::string_view("abacaba"), c);

std::cout << *x << '\n';  // UB (object destroyed)
std::cout << *y << '\n';  // Ok (string exists in global scope)
```

borrowed_range

Получается, что у одних временных объектов итераторы привисают (string, vector, list), а у других - нет (string_$view$, $span$,).

Для обозначения последних был введен дополнительный концепт - borrowed range (заимствованный диапазон).

Как отличить borrowed range от не borrowed range?

```
template <class R>
concept borrowed_range = std::ranges::range<R> &&
     (std::is_lvalue_reference_v<R> ||
      std::ranges::enable_borrowed_range<std::remove_cvref_t<R>>);

class StringView { ... };

namespace std::ranges {
template <>
inline constexpr bool enable_borrowed_range = true;
}
```

38.5.1 borrowed_iterator

Самое время посмотреть на возвращаемое значение новых алгоритмов:

```
template <std::ranges::input_range R, class T,
    class Proj = std::identity>
std::ranges::borrowed_iterator_t<R> find(R&& r,
    const T& value, Proj proj = {});
```

borrowed$_i$$terator_t < R > -, iterator_t, R - borrowed_r ange, std :: ranges :: dangling.$

```
template<std::ranges::range R>
using borrowed_iterator_t = std::conditional_t<
    std::ranges::borrowed_range<R>,
    std::ranges::iterator_t<R>, std::ranges::dangling>;
```

То есть, если диапазон возвращает хороший итератор, то borrowed$_i$$terator., std :: ranges :: dangling - ().!$

К чему это приводит?

```
// passing a temporary object, returns dangling
auto x = std::ranges::find(std::vector{1, 2, 3, 4, 5}, 3);
// passing a temporary object, returns iterator to the global scope
auto y = std::ranges::find(std::string_view("abacaba"), c);

std::cout << *x << '\n';  // CE (std::ranges::dangling)
std::cout << *y << '\n';  // Ok (string exists in global scope)
```

Теперь использование провисших итераторов не UB, а CE!

38.6 views (C++20)

С точки зрения семантики view - это легковесный объект (диапазон), который содержит представление некоторого другого диапазона.

Например, std::string_view - это диапазон, но при этом он хранит в себе всего лишь указатель на другую строчку.

Простейшими примерами view являются std::views::all и std::ranges::subrange, которые всего лишь хранят пару begin - end для некоторого другого диапазона:

```
std::vector<int> v{...};

// passing the whole vector - expensive
SomeFunction(std::views::all(v));

SomeFunction(std::ranges::subrange(v.begin() + 2, v.begin() + 1000));

auto [first, last] = std::ranges::equal_range(v, 10);
for (auto x : std::ranges::subrange(first, last)) { ... }

for (auto x : std::ranges::subrange(v.rbegin(), v.rend())) { ... }
```

38.6.1 view-генераторы (C++20)

Так как view - легковесный объект и все, что от него требуется - определить begin / end, то он может генерировать свои данные "на лету".

```
for (auto x : std::views::empty) { ... }  // empty view

for (auto x : std::views::single(1)) { ... }  // [ 1 ]

for (auto x : std::views::iota(0, 3)) { ... } // [0, 1, 2]

for (auto x : std::views::iota(0)) { ... }  // [0, 1, 2, ...]

for (auto x : std::views::repeat(3)) { ... }  // [3, 3, 3, ...]
```

38.6.2 view-адаптеры (C++20)

Адаптеры - view, которые позволяют изменить видимую часть последовательности. То есть подменить реальные элементы, теми которые требуются.

```
std::vector<int> v{ ... };
auto is_even = [](int x) { return x % 2 == 0; };

// only even values of v
for (auto x : std::views::filter(v, is_even)) { ... }
// squares of v's values
for (auto x : std::views::transform(v, [](int val) {
  return val * val; })) { ... }
// first 5 values
for (auto x : std::views::take(v, 5)) { ... }
// until an odd number is met
for (auto x : std::views::take_while(v, is_even)) { ... }
// skip the first 5 values
for (auto x : std::views::drop(v, 5)) { ... }
// skip initial even numbers
for (auto x : std::views::drop_while(v, is_even)) { ... }
// reversed sequence
for (auto x : std::views::reverse(v)) { ... }
```

Адаптеры можно комбинировать

```
std::vector<int> v{ ... };
auto is_even = [](int x) { return x % 2 == 0; };
namespace vs = std::views;
for (auto x : vs::reverse(vs::transform(vs::filter(v, is_even),
    [](int x) { return x * x; }))) { ... }
```

И даже так!

```
// sorting the first consecutive even numbers
// (excluding the first 5 numbers)
std::ranges::sort(std::views::take_while(
    std::views::drop(v, 5), is_even));
```

У адаптеров есть более удобочитаемая форма :

```
std::vector<int> v{ ... };
auto is_even = [](int x) { return x % 2 == 0; };
namespace vs = std::views;
for (auto x : vs::filter(v, is_even) |  vs::transform([](int x) {
  return x * x; }) | vs::reverse) { ... }

// sorting the first consecutive even numbers
// (excluding the first 5 numbers)
std::ranges::sort(std::views::drop(v, 5) |
std::views::take_while(is_even));
```