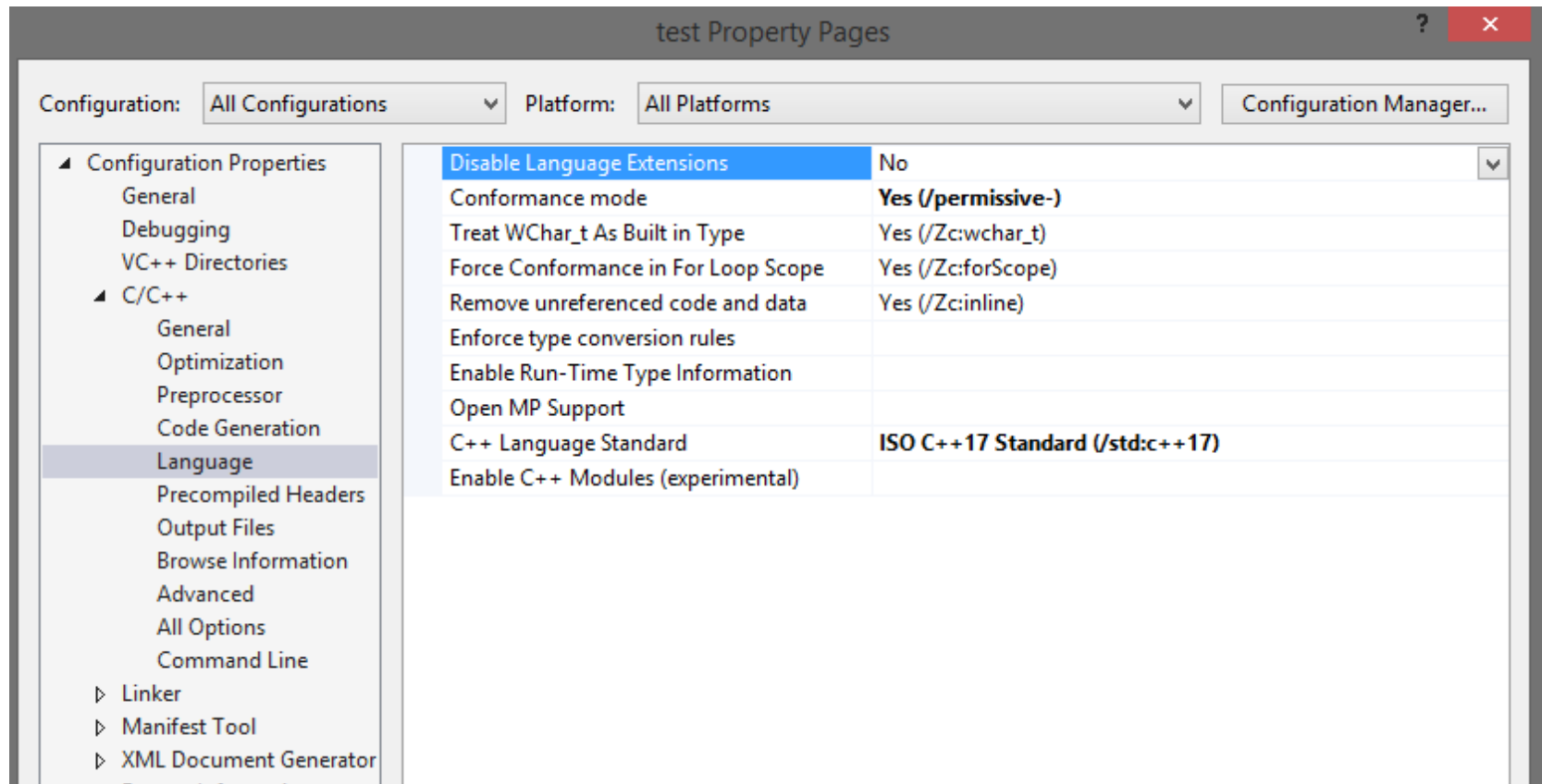


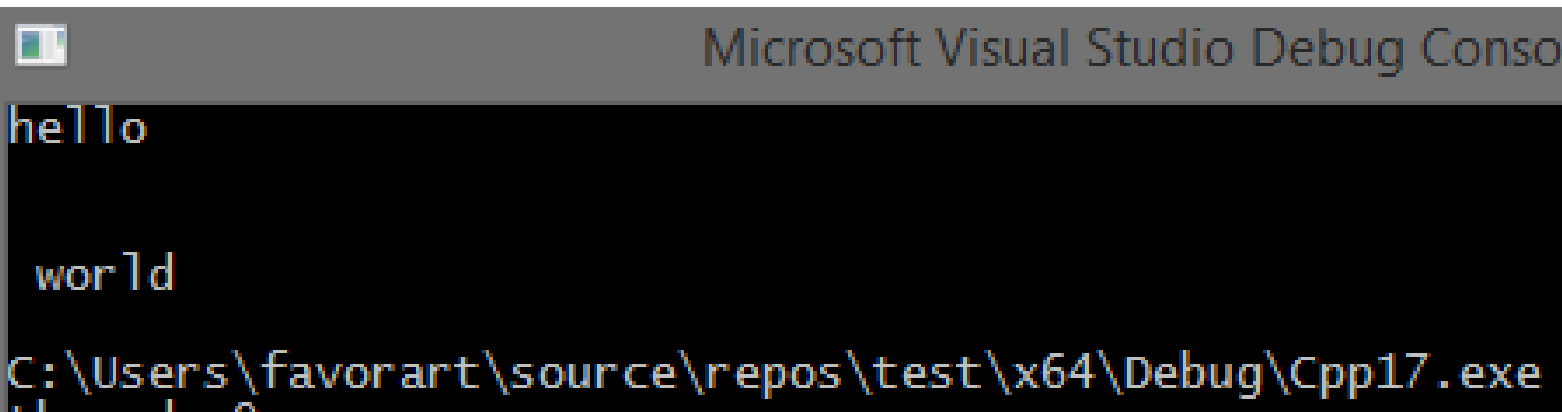
Лекция

C++17

C++17: MS VS (2017)



4 to 6



```
1. #include
2. #include
3. #include
4. using namespace std;

5. class A {
6.     string text{};
7. public:
8.     template <typename T, typename = typename enable_if_t<
9.         !is_base_of_v<A, decay_t<T>> &&
10.         !is_integral_v<remove_reference_t<T>> >>
11.     explicit A(T&& str) : text(forward<T>(str)) {
12.         cout << str << endl;
13.     }
14.     explicit A(int x) : text(to_string(x)) {}
15. };
16. int main() {
17.     string s = "hello"s;
18.     A a1(s);
19.     A a2(" world"s);
20.     A a3(string(" world"));
21.     A a4(" world");
22.     A a5(34);
23.     A a6(a2);
24.     return 0;
25. }
```

Что будет напечатано?

```
1.  #include <string>
2.  #include <iostream>
3.  #include <type_traits>
4.  using namespace std;

5.  class C {
6.      C() = delete;
7.  public:
8.      int f() { return 5; }
9.  };

10. int main()
11. {
12.     C c1{};
13.     decltype(C{}.f()) c2 = 5;           // error
14.     decltype(decltype(C{}.f()) c3 = 5;  // ok - происходит только вывод типа
15.     // работает во время компиляции и только в таких контекстах
16.     return 0;
17. }
```

Output

Show output from: Build

```
1>----- Rebuild All started: Project: Cpp17, Configuration: Debug x64 -----
1>main.cpp
1>Project1.vcxproj -> C:\Users\favorart\source\repos\test\x64\Debug\Cpp17.exe
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====
```

Что будет напечатано?

```
1.  #include <string>
2.  #include <iostream>
3.  #include <type_traits>
4.  using namespace std;

5.  class C {
6.      int i;
7.      C() = delete;
8.  public:
9.      int f() { return 5; }
10. };

11. int main()
12. {
13.     C c1{};
14.     decltype(C{}.f()) c2 = 5;           // error
15.     decltype(declval<C>().f()) c3 = 5;  // ok - происходит только вывод типа
16.     // работает во время компиляции и только в таких контекстах
17.     return 0;
18.
```

Output

Show output from: Build

1>----- Build started: Project: Cpp17, Configuration: Debug x64 -----

1>main.cpp

1>c:\users\favorart\source\repos\test\project1\main.cpp(41): error C2280: 'C::C(void)': attempting to reference a

1>c:\users\favorart\source\repos\test\project1\main.cpp(33): note: see declaration of 'C::C'

1>c:\users\favorart\source\repos\test\project1\main.cpp(33): note: 'C::C(void)': function was explicitly deleted

1>Done building project "Project1.vcxproj" -- FAILED.

===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====

НОВЫЕ ВОЗМОЖНОСТИ

- Новые методы вставки в **std::map** и **std::unordered_map**
- Унифицированный доступ к размеру контейнера **std::size()**
- Работа с файловой системой на основе **boost::filesystem**
- Параллельные версии алгоритмов STL
- Некоторые математические функции
- Большую часть экспериментальной библиотеки **TS 1**
- Новый синтаксис для распаковки
- Инициализация переменной в **if** и **switch**
- Автоматический вывод аргументов шаблона класса; также поддерживается ручное указание правил вывода.

Удалены или запрещены

- **заголовочные файлы** библиотеки Си, удалены `<ccomplex>`, `<cstdalign>`, `<cstdbool>`, `<ctgmath>`, но `<ciso646>` не запрещён
- Слово **register** осталось зарезервировано, но ничего не значит
- **операция++** для **bool**, операция **операция--** и так отсутствует
- **заявленные исключения**
 - добавляли больше вреда, чем пользы
 - `void f() throw(A, B, C)`
 - остался `throw()` как синоним `noexcept(true)`
- **повторное объявление constexpr-переменных**
 - функциональность **constexpr** аналогична новым **inline** переменным
 - `struct X {static constexpr int n = 10;}; int X::n;`
 - код явно избыточен, но почему-то был разрешён
- **типы и функции, получившие замену**
 - отдельные части **iostream**, запрещённые ещё в **C++98**
 - `std::auto_ptr` → `unique_ptr`
 - `std::random_shuffle` → `shuffle`
- **конструкторы std::function**, принимавшие аллокатор
 - из-за непонятной семантики и сложностей реализации

Удалены или запрещены

редкие возможности стандартной библиотеки

- **`std::allocator<void>`** — не востребован;
- часть функций **`allocator`** — дублируются в **`allocator_traits`**;
- **`std::raw_storage_iterator`** — не вызывает конструкторов и потому ограничен по применению;
- **`std::get_temporary_buffer`** — имеет неочевидные подводные камни;
- **`std::is_literal_type`** — бесполезен для обобщённого кода, но оставлен, пока существует понятие «*литеральный тип*»;
- **`std::iterator`** — проще писать итераторы с нуля, чем основываться на нём;
- **header `<codecvt>`** — на поверку работал очень плохо, комитет призвал пользоваться специализированными библиотеками;
- **`std::shared_ptr::unique()`** — из-за ненадёжности в многопоточной среде.

Глобальные изменения

- В сигнатуру типов функций и методов добавлена спецификация исключений
- Теперь функции разных типов
 - `void f() noexcept(true);`
 - `void f() noexcept(false);`
- но не могут формировать набор перегруженных
- API сможет требовать **callback** обработчики, которые не выбрасывают исключений

Свёртка шаблонных аргументов

Свёртка

- Шаблоны с переменным количеством параметров использовать не так и удобно:
- для развёртывания пачки параметров, приходилось идти на ухищрения:
 - рекурсивное инстанцирование и
 - откровенное «злоупотребление» разными средствами языка

Свёртка

использование аргументов функций для раскрытия пака не даёт гарантий результата

```
1.  #include <iostream>
2.  #include <string>

3.  template <bool Bit>
4.  std::string bitToString() { return Bit ? "1" : "0"; }

5.  template <typename... Args>
6.  void stub(Args&&...) {}

7.  template <bool... Bits>
8.  std::string linearBitsUnrolling() {
9.      std::string bits;
10.     stub((bits += bitToString<Bits>())....);
11.     return bits;
12. }

13. int main() {
14.     std::cout << linearBitsUnrolling<1, 0, 1, 0, 0, 0>() << "\n";
15.     return 0;
16. }
```

Свёртка

использование аргументов функций для раскрытия пака не даёт гарантий результата

```
1.  #include <iostream>
2.  #include <string>

3.  template <bool Bit>
4.  std::string bitToString() { return Bit ? "1" : "0"; }

5.  template <typename... Args>
6.  void stub(Args&&...) {}

7.  template <bool... Bits>
8.  std::string linearBitsUnrolling() {
9.      std::string bits;
10.     ((bits += bitToString())....);
11.     return bits;
12. }

13. int main() {
14.     std::cout << linearBitsUnrolling<1, 0, 1, 0, 0, 0>() << "\n";
15.     return 0;
16. }
```

C++17: выражение свёртки (fold expression)

- **Свёртка** – это функция, которая применяет заданную *комбинирующую функцию* к последовательным парам элементов в списке и возвращает результат.

Свёртка списка параметров шаблонов

- Имеет смысл только для нераскрытого пака шаблонных параметров
- Любое выражение свёртки должно быть заключено в скобки
 - ***(выражение содержащее пак аргументов)***
- Применяет на паке параметров оператор

Свёртка

суммирование элементов списка при помощи свёртки

```
1.  void f()
2.  {
3.      std::vector<int> vec = { 1, 3, 5, 7 };
4.      auto res = std::accumulate(vec.begin(),
5.                                  vec.end(),
6.                                  0 /* init val */,
7.                                  [](const int &a, const int &b) -> int {
8.                                      return a + b;
9.                                  });
10.     std::cout << res << '\n';
11. }

12. // 1 + (3 + (5 + (7 + 0))) = 16
```

Свёртка параметров шаблона (fold expressions)

правое унарное выражение свёртки

```
1.  #include <iostream>

2.  template <int... values>
3.  constexpr int rightSum() {
4.      return (values + ...);
5.  }

6.  int main() {
7.      std::cout << rightSum<1, 2, 3, 4, 5>() << "\n";
8.      return 0;
9.  }
```

левое унарное выражение свёртки

```
10. template <int... values>
11. constexpr int leftSum() {
12.     return (... + values);
13. }
```

Свёртка параметров шаблона (fold expressions)

(pack op ...)	Унарная правоассоциативная свертка
(... op pack)	Унарная левоассоциативная свертка
(pack op ... op init)	Бинарная правоассоциативная свертка
(init op ... op pack)	Бинарная левоассоциативная свертка

+ - * / % ^ & | ~ = < > << >> += -= *= /= %= ^=
&= |= <<= >>= == != <= >= && || , .* ->*

Свёртка параметров шаблона (fold expressions)

бинарная свёртка

```
1.  // C++17
2.  template<typename... Args>
3.  auto BinaryFold(Args... args) {
4.      return (args + ... + 100);
5.  }
6.  int main() {
7.      std::cout << BinaryFold(1, 2, 3, 4, 5) << '\n'; // 115
8.      return 0;
9.  }
```

оператор ',' (запятая)

раскроет `pack` в последовательность действий

```
((bits += bitToString<Bits>()))...);
```

```
1.  template<typename T, typename... Args>
2.  void PushToVector(std::vector<T>& v, Args&&... args)
3.  {
4.      (v.push_back(std::forward<Args>(args)), ...);
5.      // Раскрывается в последовательность выражений через запятую вида:
6.      // v.push_back(std::forward<Args_1>(arg1)),
7.      // v.push_back(std::forward<Args_2>(arg2)),
8.      // ...
9.  }

10. int main() {
11.     std::vector<int> vec;
12.     PushToVector(vec, 1, 4, 5, 8);
13.     return 0;
14. }
```

Свёртка параметров шаблона (fold expressions)

бинарная свёртка

```
1.  // C++17
2.  template<typename... Args>
3.  auto BinaryFold(Args... args) {
4.      return (args + ... + 100);
5.  }
6.  int main() {
7.      std::cout << BinaryFold(1, 2, 3, 4, 5) << '\n'; // 115
8.      return 0;
9.  }
```

нужно явно указывать правила для рекурсии

```
10. // C++14
11. auto Sum() {
12.     return 0;
13. }
14. template<typename Arg, typename... Args>
15. auto Sum(Arg first, Args... rest) {
16.     return first + Sum(rest...);
17. }
18. int main() {
19.     std::cout << Sum(1, 2, 3, 4); // 10
20.     return 0;
21. }
```

Свёртка параметров шаблона (fold expressions)

унарные выражения

1. `(PPack op ...)`
2. `(... op PPack)`

Свёртка параметров шаблона (fold expressions)

унарные выражения

1. $(\text{PPack op } \dots) = \text{PP}_1 \text{ op } (\text{PP}_2 \text{ op } \dots (\text{PP}_{N-2} \text{ op } (\text{PP}_{N-1})) \dots)$ /* правое */
2. $(\dots \text{ op PPack}) = (((\text{PP}_1) \text{ op PP}_2) \text{ op PP}_3) \text{ op PP}_4 \dots$ /* левое */

Свёртка параметров шаблона (fold expressions)

бинарные выражения

1. $(\text{PPack op } \dots) = \text{PP}_1 \text{ op } (\text{PP}_2 \text{ op } \dots (\text{PP}_{N-2} \text{ op } (\text{PP}_{N-1})) \dots)$ /* правое */
2. $(\dots \text{ op PPack}) = (((\text{PP}_1) \text{ op PP}_2) \text{ op PP}_3) \text{ op PP}_4) \dots$ /* левое */
3. **(PPack op ... op X)**
4. **(X op ... op PPack)**

Свёртка параметров шаблона (fold expressions)

бинарные выражения

1. $(\text{PPack op } \dots) = \text{PP}_1 \text{ op } (\text{PP}_2 \text{ op } \dots (\text{PP}_{N-2} \text{ op } (\text{PP}_{N-1})) \dots)$ /* правое */
2. $(\dots \text{ op PPack}) = (((\text{PP}_1) \text{ op PP}_2) \text{ op PP}_3) \text{ op PP}_4) \dots$ /* левое */
3. $(\text{PPack op } \dots \text{ op X}) = \text{PP}_1 \text{ op } (\text{PP}_2 \text{ op } \dots (\text{PP}_{N-1} \text{ op X}) \dots)$ /* правое */
4. $(\text{X op } \dots \text{ op PPack}) = (((\text{X op PP}_1) \text{ op PP}_2) \text{ op PP}_3) \dots$ /* левое */

Свёртка параметров шаблона (fold expressions)

бинарные выражения

1. `(PPack op ...) = PP1 op (PP2 op ... (PPN-2 op (PPN-1))...)` */* правое */*
2. `(... op PPack) = (((PP1) op PP2) op PP3) op PP4) ...` */* левое */*
3. `(PPack op ... op X) = PP1 op (PP2 op ... (PPN-1 op X)...))` */* правое */*
4. `(X op ... op PPack) = ((X op PP1) op PP2) op PP3)...` */* левое */*
5. `args = { 1., 9., 4., 7. }`
6. `(args += ...)`
7. `(... += args)`
8. `(117 - ... - args)`
9. `(args *= ... *= 10)`

Свёртка параметров шаблона (fold expressions)

бинарные выражения

1. `(PPack op ...) = PP1 op (PP2 op ... (PPN-2 op (PPN-1))...)` */* правое */*
2. `(... op PPack) = (((PP1) op PP2) op PP3) op PP4) ...` */* левое */*
3. `(PPack op ... op X) = PP1 op (PP2 op ... (PPN-1 op X)...))` */* правое */*
4. `(X op ... op PPack) = ((X op PP1) op PP2) op PP3)...` */* левое */*
5. `args = { 1., 9., 4., 7. }`
6. `(args += ...) = 1 += (9 += (4 += (7))) = { 21, 20, 11, 7 }`
7. `(... += args) = ((1) += 9) += 4) += 7 = { 1, 10, 14, 21 }`
8. `(117 - ... - args) = (((117 - 1) - 9) - 4) - 7 = { 116, 107, 103, 96 }`
9. `(args * ... * 10) = 1 * (9 * (4 * (7 * 10))) = { 2520, 2520, 280, 70 }`

Свёртка параметров шаблона (fold expressions)

бинарные выражения

1. `(PPack op ...) = PP1 op (PP2 op ... (PPN-2 op (PPN-1))...)` */* правое */*
2. `(... op PPack) = (((PP1) op PP2) op PP3) op PP4) ...` */* левое */*
3. `(PPack op ... op X) = PP1 op (PP2 op ... (PPN-1 op X)...))` */* правое */*
4. `(X op ... op PPack) = ((X op PP1) op PP2) op PP3)...` */* левое */*
5. `args = { 1., 9., 4., 7. }`
6. `(args += ...) = 1 += (9 += (4 += (7))) = { 21, 20, 11, 7 }`
7. `(... += args) = ((1) += 9) += 4) += 7 = { 1, 10, 14, 21 }`
8. `(117 - ... - args) = (((117 - 1) - 9) - 4) - 7 = { 116, 107, 103, 96 }`
9. `(args * ... * 10) = 1 * (9 * (4 * (7 * 10))) = { 2520, 2520, 280, 70 }`

Свёртка параметров шаблона (fold expressions)

что делать, если пачка параметров пустая?

1. **правила:**

2. пустая пачка параметров заменяется на предопределённое значение, в зависимости от оператора свёртки

3. **Оператор Значение**

4. && true

5. || false

6. , void()

7. с большинством операторов пустая пачка **работать не будет**

Раскрытие пачки и using

шаблоны с переменным количеством параметров и статическая рекурсия

```
1. struct IntPrinter {
2.     void operator()(int i) { cout << "Int passed: " << i << "\n"; }
3. };
4. struct FloatPrinter {
5.     void operator()(float f) { cout << "Float passed: " << f << "\n"; }
6. };

7. int main() {
8.     Printer<IntPrinter, FloatPrinter> printer;
9.     printer(55);
10.    printer(7.1f);
11.    printer17(19);
12.    printer17(0.081f);
13.    return 0;
14. }
```

Раскрытие пачки и using

шаблоны с переменным количеством параметров и статическая рекурсия

```
1.  // C++14
2.  template<typename... Ts>
3.  struct Printer;

4.  template<typename T, typename... Ts>
5.  struct Printer<T, Ts...> : public T, Printer<Ts...> {
6.      using T::operator();
7.      using Printer<Ts...>::operator();
8.  };

9.  template <>
10. struct Printer<> {
11.     void operator()(); // !!!
12. };

```

Раскрытие пачки и using

предложение ([P0195R2](#))

```
1.  // C++14
2.  template<typename... Ts>
3.  struct Printer;

4.  template<typename T, typename... Ts>
5.  struct Printer<T, Ts...> : public T, Printer<Ts...> {
6.      using T::operator();
7.      using Printer<Ts...>::operator();
8.  };

9.  template <>
10. struct Printer<> {
11.     void operator()(); // !!!
12. };

13. // C++17
14. template<typename... Ts>
15. struct Printer : public Ts... {
16.     using Ts::operator()...;

17. };
```

Раскрытие пачки и using

предложение ([P0195R2](#))

```
1.  // C++14
2.  template<typename... Ts>
3.  struct Printer;

4.  template<typename T, typename... Ts>
5.  struct Printer<T, Ts...> : public T, Printer<Ts...> {
6.      using T::operator();
7.      using Printer<Ts...>::operator();
8.  };

9.  template <>
10. struct Printer<> {
11.     void operator()(); // !!!
12. };

13. // C++17
14. template<typename... Ts>
15. struct Printer : public Ts... {
16.     using Ts::operator()...;
17.     using IntPrinter::operator(), FloatPrinter::operator();
18. };
```

Выведение типов

Нетипичные шаблоны

non-type template - шаблоны, независящие от типа

```
1.  // C++17
2.  template<auto constant>
3.  void NonTypeTemplateFunc() {
4.      std::cout << "Constant provided: " << constant << "\n";
5.  }

6.  // C++14
7.  template<typename Type, Type constant>
8.  void NonTypeTemplateFunc() {
9.      std::cout << "Constant provided: " << constant << "\n";
10. }

11. int main() {
12.     NonTypeTemplateFunc<int, 42>();
13.     NonTypeTemplateFunc<char, 'c'>();
14.     return 0;
15. }
```

Вывод типов шаблонных параметров

не указывать аргументы шаблонов

```
1.  #include <tuple>
2.  #include <array>
3.  template<typename T, typename U>
4.  struct S {
5.      T m_first;
6.      U m_second;
7.      S(T first, U second) : m_first(first), m_second(second) {}
8.  };

9.  int main() {
10.     // C++14
11.     auto p = std::make_pair(10, 'c');
12.     std::pair<char, int> p1 = { 'c', 42 };
13.     std::tuple<char, int, double> t1 = { 'c', 42, 3.14 };
14.     S<int, char> s1 = { 10, 'c' };

15.     // C++17
16.     std::pair p2 = { 'c', 42 };
17.     std::tuple t2 = { 'c', 42, 3.14 };
18.     S s2 = { 10, 'c' };
19.     return 0;
20. }
```

Вывод типов шаблонных параметров

```
1.  void main() {
2.      std::vector<int> vector;
3.      int i = 5;
4.      auto begin = vector.begin();
5.      auto end = vector.end();
6.      std::string string{ "me" };

7.      // C++14
8.      std::tuple<std::vector<int>,
9.                int,
10.               std::vector<int>::iterator,
11.               std::vector<int>::iterator,
12.               std::string> tuple{ vector, i, begin, end, string };
13.      std::sort(vector.begin(), vector.end(), std::greater<int>());

14.     // C++17
15.     auto tuple = std::make_tuple(vector, i, begin, end, string);
16.     std::array arr = { 1,2,3,4 };

17.     std::mutex m;
18.     std::lock_guard lock{ m };

19.     std::sort(vector.begin(), vector.end(), std::greater{});
20. }
```

Вывод типов шаблонных параметров

определено множество правил вывода типов (deduction guides)

```
1.  #include <iostream>

2.  template<typename T, typename U>
3.  struct S {
4.      T m_first;
5.      U m_second;
6.  };

7.  // C++17 - deduction guide
8.  template<typename T, typename U>
9.  S(const T &first, const U &second) -> S<T,U>;

10. int main() {
11.     S s = { 42, "hello" };
12.     std::cout << s.m_first << s.m_second << '\n';
13.     return 0;
14. }
```

Вывод типов шаблонных параметров

Выведение аргументов классов-шаблонов
[over.match.class.deduct]:

1. Для каждого конструктора исходного класса-шаблона (только основной (**primary**) шаблон, не специализации) генерируется *шаблонная функция*, **параметры шаблона которой** есть **параметры класса-шаблона**, а **аргументы функции** есть **аргументы конструктора**. **Возвращаемое значение функции** есть **исходный шаблонный класс** с соответствующими аргументами шаблона.
2. Вне зависимости от того, какие конструкторы есть у класса, **добавляется** ещё одна **функция**, выведенная из гипотетического конструктора вида **T(T)**.
3. Полученные ранее *функции* становятся **конструкторами гипотетического нешаблонного класса**, посредством создания объекта которого будет производиться конечное определение аргументов шаблона.

Вывод типов шаблонных параметров

генерируется шаблонная функция, параметры шаблона которой -параметры класса-шаблона

```
1.  template <typename T> class DummyVector {  
2.      std::vector<T> m_Storage;  
3.  };
```

Вывод типов шаблонных параметров

генерируется шаблонная функция, параметры шаблона которой -параметры класса-шаблона

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  };

4.  // Шаг 1. 3 гипотетических функции (известные конструкторы)
5.  template <typename T>
6.  DummyVector<T> DummyVector();
7.  template <typename T>
8.  DummyVector<T> DummyVector(const DummyVector<T>&);
9.  template <typename T>
10. DummyVector<T> DummyVector(DummyVector<T>&&);
```

Вывод типов шаблонных параметров

генерируется шаблонная функция, параметры шаблона которой -параметры класса-шаблона

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  };

4.  // Шаг 1. 3 гипотетических функции (известные конструкторы)
5.  template <typename T>
6.  DummyVector<T> DummyVector();
7.  template <typename T>
8.  DummyVector<T> DummyVector(const DummyVector<T>&);
9.  template <typename T>
10. DummyVector<T> DummyVector(DummyVector<T>&&);

11. // Шаг 2. Добавим ещё одну функцию
12. template <typename T>
13. DummyVector<T> DummyVector(DummyVector<T>);
```

Вывод типов шаблонных параметров

генерируется шаблонная функция, параметры шаблона которой -параметры класса-шаблона

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  };

4.  // Шаг 1. 3 гипотетических функции (известные конструкторы)
5.  template <typename T>
6.  DummyVector<T> DummyVector();
7.  template <typename T>
8.  DummyVector<T> DummyVector(const DummyVector<T>&);
9.  template <typename T>
10. DummyVector<T> DummyVector(DummyVector<T>&&);

11. // Шаг 2. Добавим ещё одну функцию
12. template <typename T>
13. DummyVector<T> DummyVector(DummyVector<T>);

14. // Шаг 3. Делает из этих функций конструкторы гипотетического класса
15. class HypotheticalDummyVector {
16. public:
17.     template <typename T> HypotheticalDummyVector();
18.     template <typename T> HypotheticalDummyVector(const DummyVector<T>&);
19.     template <typename T> HypotheticalDummyVector(DummyVector<T>&&);
20.     template <typename T> HypotheticalDummyVector(DummyVector<T>);
21. };
```

Вывод типов шаблонных параметров

примеры гипотетической связи

1. `// Предположим`
2. `DummyVector copyInit = DummyVector<int>{};`

Вывод типов шаблонных параметров

примеры гипотетической связи

```
1. // Предположим
2. DummyVector copyInit = DummyVector<int>{};
3. // Сначала создание
4. HypotheticalDummyVector hdv{ DummyVector<int>{} };
```

Вывод типов шаблонных параметров

примеры гипотетической связи

```
1.  // Предположим
2.  DummyVector copyInit = DummyVector<int>{};
3.  // Сначала создание
4.  HypotheticalDummyVector hdv{ DummyVector<int>{} };
5.  // "код не соберётся, тут явная двусмысленность!"
```

Вывод типов шаблонных параметров

примеры гипотетической связи

```
1.  // Предположим
2.  DummyVector copyInit = DummyVector<int>{};
3.  // Сначала создание
4.  HypotheticalDummyVector hdv{ DummyVector<int>{} };
5.  // "код не соберётся, тут явная двусмысленность!"
6.  // у гипотетических классов правила перегрузки свои, однозначно выбрав конструктор
7.  HypotheticalDummyVector(DummyVector<T>);
```

Вывод типов шаблонных параметров

примеры гипотетической связи

```
1.  // Предположим
2.  DummyVector copyInit = DummyVector<int>{};
3.  // Сначала создание
4.  HypotheticalDummyVector hdv{ DummyVector<int>{} };
5.  // "код не соберётся, тут явная двусмысленность!"
6.  // у гипотетических классов правила перегрузки свои, однозначно выбрав конструктор
7.  HypotheticalDummyVector(DummyVector<T>);      // поняв, что int → T
8.  // «активируется» гипотетическая связь конструктора с соответствующей функцией
9.  DummyVector<T> DummyVector(DummyVector<T>);
```

Вывод типов шаблонных параметров

примеры гипотетической связи

```
1.  // Предположим
2.  DummyVector copyInit = DummyVector<int>{};
3.  // Сначала создание
4.  HypotheticalDummyVector hdv{ DummyVector<int>{} };
5.  // "код не соберётся, тут явная двусмысленность!"
6.  // у гипотетических классов правила перегрузки свои, однозначно выбрав конструктор
7.  HypotheticalDummyVector(DummyVector<T>);      // поняв, что int → T
8.  // «активируется» гипотетическая связь конструктора с соответствующей функцией
9.  DummyVector<T> DummyVector(DummyVector<T>);

10. template <typename T> class DummyVector {
11.     std::vector<T> m_Storage;
12. public:
13.     DummyVector<T> DummyVector(std::initializer_list<T> lst) : m_Storage(lst) {}
14.     // ...
15. };
```

Вывод типов шаблонных параметров

примеры гипотетической связи

```
1.  // Предположим
2.  DummyVector copyInit = DummyVector<int>{};
3.  // Сначала создание
4.  HypotheticalDummyVector hdv{ DummyVector<int>{} };
5.  // "код не соберётся, тут явная двусмысленность!"
6.  // у гипотетических классов правила перегрузки свои, однозначно выбрав конструктор
7.  HypotheticalDummyVector(DummyVector<T>);          // поняв, что int → T
8.  // «активируется» гипотетическая связь конструктора с соответствующей функцией
9.  DummyVector<T> DummyVector(DummyVector<T>);

10. template <typename T> class DummyVector {
11.     std::vector<T> m_Storage;
12. public:
13.     DummyVector<T> DummyVector(std::initializer_list<T> lst) : m_Storage(lst) {}
14.     // ...
15. };
16. class HypotheticalDummyVector {
17. public:
18.     // ... // Шаги 1-3 добавят конструктор
19.     template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst);
20. };
```

Вывод типов шаблонных параметров

примеры гипотетической связи

```
1.  // Предположим
2.  DummyVector copyInit = DummyVector<int>{};
3.  // Сначала создание
4.  HypotheticalDummyVector hdv{ DummyVector<int>{} };
5.  // "код не соберётся, тут явная двусмысленность!"
6.  // у гипотетических классов правила перегрузки свои, однозначно выбрав конструктор
7.  HypotheticalDummyVector(DummyVector<T>);      // поняв, что int → T
8.  // «активируется» гипотетическая связь конструктора с соответствующей функцией
9.  DummyVector<T> DummyVector(DummyVector<T>);

10. template <typename T> class DummyVector {
11.     std::vector<T> m_Storage;
12. public:
13.     DummyVector<T> DummyVector(std::initializer_list<T> lst) : m_Storage(lst) {}
14.     // ...
15. };
16. class HypotheticalDummyVector {
17. public:
18.     // ... // Шаги 1-3 добавят конструктор
19.     template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst);
20. };
21. // ...
22. DummyVector initList{5.0, .3, .1, 5.67};
```

Вывод типов шаблонных параметров

двигаемся к сути

```
1.  template <typename T> class DummyVector {  
2.      std::vector<T> m_Storage;  
3.  public:  
4.      template<typename Iter>  
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}  
6.  };  
  
7.  std::vector<int> vector{ 1, 2, 3, 4, 5 };  
8.  DummyVector initFromVec{ vector.begin(), vector.end() };
```

Вывод типов шаблонных параметров

двигаемся к сути

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  public:
4.      template<typename Iter>
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}
6.  };

7.  std::vector<int> vector{ 1, 2, 3, 4, 5 };
8.  DummyVector initFromVec{ vector.begin(), vector.end() };
9.  // пример никак не соберётся - никаких указаний на то, откуда можно вывести тип T
10. // в конструкторе есть тип Iter, но нет T
```

Вывод типов шаблонных параметров

двигаемся к сути

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  public:
4.      template<typename Iter>
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}
6.  };

7.  std::vector<int> vector{ 1, 2, 3, 4, 5 };
8.  DummyVector initFromVec{ vector.begin(), vector.end() };
9.  // пример никак не соберётся - никаких указаний на то, откуда можно вывести тип T
10. // в конструкторе есть тип Iter, но нет T
11. // нужна сущность нового стандарта - инструкция выведения [типа] (deduction guide)
12. // инструкции должны находиться в том же уровне доступа, что и класс-шаблон
13. // (:: или namespaces)
```

Вывод типов шаблонных параметров

двигаемся к сути

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  public:
4.      template<typename Iter>
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}
6.  };

7.  std::vector<int> vector{ 1, 2, 3, 4, 5 };
8.  DummyVector initFromVec{ vector.begin(), vector.end() };
9.  // пример никак не соберётся - никаких указаний на то, откуда можно вывести тип T
10. // в конструкторе есть тип Iter, но нет T
11. // нужна сущность нового стандарта - инструкция выведения [типа] (deduction guide)
12. // инструкции должны находиться в том же уровне доступа, что и класс-шаблон
13. // (:: или namespaces)

14. template <typename T>
15. DummyVector(const DummyVector<T>&) -> DummyVector<T>;
16. // инструкция выведения типа из его копии,
```

Вывод типов шаблонных параметров

двигаемся к сути

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  public:
4.      template<typename Iter>
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}
6.  };

7.  std::vector<int> vector{ 1, 2, 3, 4, 5 };
8.  DummyVector initFromVec{ vector.begin(), vector.end() };
9.  // пример никак не соберётся - никаких указаний на то, откуда можно вывести тип T
10. // в конструкторе есть тип Iter, но нет T
11. // нужна сущность нового стандарта - инструкция выведения [типа] (deduction guide)
12. // инструкции должны находиться в том же уровне доступа, что и класс-шаблон
13. // (:: или namespaces)

14. template <typename T>
15. DummyVector(const DummyVector<T>&) -> DummyVector<T>;
16. // инструкция выведения типа из его копии,

17. // может ли быть так?
18. DummyVector<double> inst = DummyVector<int>{}; // compile error
```

Вывод типов шаблонных параметров

двигаемся к сути

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  public:
4.      template<typename Iter>
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}
6.  };

7.  std::vector<int> vector{ 1, 2, 3, 4, 5 };
8.  DummyVector initFromVec{ vector.begin(), vector.end() };
9.  // пример никак не соберётся - никаких указаний на то, откуда можно вывести тип T
10. // в конструкторе есть тип Iter, но нет T
11. // нужна сущность нового стандарта - инструкция выведения [типа] (deduction guide)
12. // инструкции должны находиться в том же уровне доступа, что и класс-шаблон
13. // (:: или namespaces)

14. template <typename T>
15. DummyVector(const DummyVector<T>&) -> DummyVector<T>;
16. // инструкция выведения типа из его копии,
17. // для конструктора тип результата фиксировано, а параметры шаблона по усмотрению
18. template <typename T>
19. DummyVector(const DummyVector<T>&) -> DummyVector<double>;
20. // может ли быть так?
21. DummyVector<double> inst = DummyVector<int>{}; // compile error
```

Вывод типов шаблонных параметров

двигаемся к сути

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  public:
4.      template<typename Iter>
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}
6.  };

7.  std::vector<int> vector{ 1, 2, 3, 4, 5 };
8.  DummyVector initFromVec{ vector.begin(), vector.end() };

9.  template<typename Iter> DummyVector(Iter begin, Iter end) ->
10.     DummyVector<typename std::iterator_traits<Iter>::value_type>;
```

Вывод типов шаблонных параметров

инструкции вывода типа в общем шаблоне автоматического вывода

```
1.  template <typename T> class DummyVector {
2.      std::vector<T> m_Storage;
3.  public:
4.      template<typename Iter>
5.      DummyVector(Iter begin, Iter end) : m_Storage(begin, end) {}
6.      DummyVector(std::initializer_list<T> lst) : m_Storage(lst) {}
7.  };

8.  template<typename Iter> DummyVector(Iter begin, Iter end) ->
9.      DummyVector<typename std::iterator_traits<Iter>::value_type>;

10. class HypotheticalDummyVector {
11. public:
12.     template <typename T> HypotheticalDummyVector(const DummyVector<T>&);           // #1
13.     template <typename T> HypotheticalDummyVector(DummyVector<T>&&);                 // #2
14.     template <typename T> HypotheticalDummyVector(DummyVector<T>);                 // #3
15.     template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst);    // #4
16.     template <typename T, typename It> HypotheticalDummyVector(It begin, It end);  // #5
17. };

```

Вывод типов шаблонных параметров

правила перегрузки гипо-класса

```
1.  class HypotheticalDummyVector {
2.  public:
3.      template <typename T> HypotheticalDummyVector(const DummyVector<T>&);           // #1
4.      template <typename T> HypotheticalDummyVector(DummyVector<T>&&);               // #2
5.      template <typename T> HypotheticalDummyVector(DummyVector<T>);                 // #3
6.      template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst);    // #4
7.      template <typename T, typename It> HypotheticalDummyVector(It begin, It end);    // #5
8.  };
9.  HypotheticalDummyVector hypothetical{ DummyVector<int>{} };
10. RealDummy real{ DummyVector<int>{} };

11. HypotheticalDummyVector hypothetical{ 1 };
12. RealDummy real{ 1 };
```

Вывод типов шаблонных параметров

правила перегрузки гипо-класса

```
1.  class HypotheticalDummyVector {
2.  public:
3.      template <typename T> HypotheticalDummyVector(const DummyVector<T>&);           // #1
4.      template <typename T> HypotheticalDummyVector(DummyVector<T>&&);               // #2
5.      template <typename T> HypotheticalDummyVector(DummyVector<T>);               // #3
6.      template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst); // #4
7.      template <typename T, typename It> HypotheticalDummyVector(It begin, It end); // #5
8.  };
9.  HypotheticalDummyVector hypothetical{ DummyVector<int>{} }; // вызовет #3
10. RealDummy real{ DummyVector<int>{} }; // вызовет #4
11. HypotheticalDummyVector hypothetical{ 1 }; // вызовет #4
12. RealDummy real{ 1 }; // вызовет #4
```

Вывод типов шаблонных параметров

правила перегрузки гипо-класса

```
1.  class HypotheticalDummyVector {
2.  public:
3.      template <typename T> HypotheticalDummyVector(const DummyVector<T>&);           // #1
4.      template <typename T> HypotheticalDummyVector(DummyVector<T>&&);               // #2
5.      template <typename T> HypotheticalDummyVector(DummyVector<T>);               // #3
6.      template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst); // #4
7.      template <typename T, typename It> HypotheticalDummyVector(It begin, It end); // #5
8.  };
9.  HypotheticalDummyVector hypothetical{ DummyVector<int>{} }; // вызовет #3
10. RealDummy real{ DummyVector<int>{} }; // вызовет #4

11. HypotheticalDummyVector hypothetical{ 1 }; // вызовет #4
12. RealDummy real{ 1 }; // вызовет #4
13. Специальное правило [over.match.class.deduct]
14. если есть инициализация, подходящая для конструктора с initializer_list, но
15. она состоит лишь из одного элемента, который является специализацией гипо-класса,
16. тогда правило, в первую очередь рассматривать конструктор с initializer_list,
17. перестает работать.
```

Вывод типов шаблонных параметров

правила перегрузки гипо-класса

1. `class HypotheticalDummyVector {`
2. `public:`
3. `template <typename T> HypotheticalDummyVector(const DummyVector<T>&);` // #1
4. `template <typename T> HypotheticalDummyVector(DummyVector<T>&&);` // #2
5. `template <typename T> HypotheticalDummyVector(DummyVector<T>);` // #3
6. `template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst);` // #4
7. `template <typename T, typename It> HypotheticalDummyVector(It begin, It end);` // #5
8. `};`
9. `HypotheticalDummyVector hypothetical{ DummyVector<int>{} };` // вызовет #3
10. `RealDummy real{ DummyVector<int>{} };` // вызовет #4
11. `HypotheticalDummyVector hypothetical{ 1 };` // вызовет #4
12. `RealDummy real{ 1 };` // вызовет #4
13. **Специальное правило `[over.match.class.deduct]`**
14. **Исключение `[over.match.best]`**
15. *Гипо-функция сформированная на шаге 2 побеждает в перегрузке*
16. *все функции, сгенерированные из конструкторов на шаге 1*

Вывод типов шаблонных параметров

правила перегрузки гипо-класса

```
1.  class HypotheticalDummyVector {
2.  public:
3.      template <typename T> HypotheticalDummyVector(const DummyVector<T>&);           // #1
4.      template <typename T> HypotheticalDummyVector(DummyVector<T>&&);               // #2
5.      template <typename T> HypotheticalDummyVector(DummyVector<T>);               // #3
6.      template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst); // #4
7.      template <typename T, typename It> HypotheticalDummyVector(It begin, It end); // #5
8.  };
9.  HypotheticalDummyVector hypothetical{ DummyVector<int>{} }; // вызовет #3
10. RealDummy real{ DummyVector<int>{} }; // вызовет #4
11. HypotheticalDummyVector hypothetical{ 1 }; // вызовет #4
12. RealDummy real{ 1 }; // вызовет #4
13. Специальное правило [over.match.class.deduct]
14. Исключение [over.match.best]
15. DummyVector initFromVec(vector.begin(), vector.end());
```

Вывод типов шаблонных параметров

правила перегрузки гипо-класса

```
1.  class HypotheticalDummyVector {
2.  public:
3.      template <typename T> HypotheticalDummyVector(const DummyVector<T>&);           // #1
4.      template <typename T> HypotheticalDummyVector(DummyVector<T>&&);                 // #2
5.      template <typename T> HypotheticalDummyVector(DummyVector<T>);                 // #3
6.      template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst);    // #4
7.      template <typename T, typename It> HypotheticalDummyVector(It begin, It end);    // #5
8.  };
9.  HypotheticalDummyVector hypothetical{ DummyVector<int>{} }; // вызовет #3
10. RealDummy real{ DummyVector<int>{} }; // вызовет #4
11. HypotheticalDummyVector hypothetical{ 1 }; // вызовет #4
12. RealDummy real{ 1 }; // вызовет #4
13. Специальное правило [over.match.class.deduct]
14. Исключение [over.match.best]
15. DummyVector initFromVec(vector.begin(), vector.end());
16. Новая семантика
17. гипо-функции, производные от инструкций вывода типа,
    побеждают в перегрузке все другие гипофункции.
```

Вывод типов шаблонных параметров

правила перегрузки гипо-класса

1. `class HypotheticalDummyVector {`
2. `public:`
3. `template <typename T> HypotheticalDummyVector(const DummyVector<T>&);` // #1
4. `template <typename T> HypotheticalDummyVector(DummyVector<T>&&);` // #2
5. `template <typename T> HypotheticalDummyVector(DummyVector<T>);` // #3
6. `template <typename T> HypotheticalDummyVector(std::initializer_list<T> lst);` // #4
7. `template <typename T, typename It> HypotheticalDummyVector(It begin, It end);` // #5
8. `};`
9. `HypotheticalDummyVector hypothetical{ DummyVector<int>{} };` // вызовет #3
10. `RealDummy real{ DummyVector<int>{} };` // вызовет #4
11. `HypotheticalDummyVector hypothetical{ 1 };` // вызовет #4
12. `RealDummy real{ 1 };` // вызовет #4
13. **Специальное правило [over.match.class.deduct]**
14. **Исключение [over.match.best]**
15. `DummyVector initFromVec(vector.begin(), vector.end());`
16. **Новая семантика**
17. *гипо-функции, производные от инструкций вывода типа, побеждают в перегрузке все другие гипофункции.*
18. **Семантика, аналогичная конструкторам обычных классов**
19. *если конструктор гипо-класса является производным от нешаблонного конструктора оригинала, тогда такой конструктор имеет преимущество над конструктором, производным от шаблонного конструктора гипо-класса*

Вывод типов шаблонных параметров

предсказуемый результат ценой ещё одного исключения

```
1.  template <typename T> class Dummy {
2.  public:
3.      template <typename U> Dummy(T&& t, U&& u);
4.  };

5.  class HypotheticalDummy {
6.  public:
7.      // ...
8.      template <typename T, typename U> HypotheticalDummy(T&& t, U&& u) {}
9.  };

10. int value = 0;
11. Dummy dummy{ value, 1 };
```

Вывод типов шаблонных параметров

предсказуемый результат ценой ещё одного исключения

```
1.  template <typename T> class Dummy {
2.  public:
3.      template <typename U> Dummy(T&& t, U&& u);
4.  };

5.  class HypotheticalDummy {
6.  public:
7.      // ...
8.      template <typename T, typename U> HypotheticalDummy(T&& t, U&& u) {}
9.  };

10. int value = 0;
11. Dummy dummy{ value, 1 };

12. Dummy<int, int>(int&&, int&&) lvalue
```

Вывод типов шаблонных параметров

предсказуемый результат ценой ещё одного исключения

```
1.  template <typename T> class Dummy {
2.  public:
3.      template <typename U> Dummy(T&& t, U&& u);
4.  };

5.  class HypotheticalDummy {
6.  public:
7.      // ...
8.      template <typename T, typename U> HypotheticalDummy(T&& t, U&& u) {}
9.  };

10. template <typename T, typename U> Dummy(T&&, U&&) -> Dummy<T>;
11. // без такой инструкции вывода, код не скомпилируется:
12. int value = 0;
13. Dummy dummy{ value, 1 };

14. Dummy<int, int>(int&&, int&&) lvalue
15. Dummy<int, int>(int&, int&&) lvalue
```

Вывод типов шаблонных параметров

яснее ясного

1. `template <typename T, typename U> explicit Dummy(T&&, U&&) -> Dummy<T>;`
2. `// исходные explicit сущности`
3. `// получают в гипо-классе соответствующие explicit конструкторы`

Вывод типов шаблонных параметров

яснее ясного

```
1.  template <typename T, typename U> explicit Dummy(T&&, U&&) -> Dummy<T>;
2.  // исходные explicit сущности
3.  // получают в гипо-классе соответствующие explicit конструкторы

4.  int value = 0;
5.  // оставив инструкцию вывода типа с explicit

6.  // можно разрешить
7.  Dummy dummy{ value, 1 };
8.  // и запретить
9.  Dummy dummy = { value, 1 };
10. // (конструктор гипо-класса стал explicit)
```

Вывод типов шаблонных параметров

яснее ясного

```
1.  template <typename T, typename U> explicit Dummy(T&&, U&&) -> Dummy<T>;
2.  // исходные explicit сущности
3.  // получают в гипо-классе соответствующие explicit конструкторы

4.  int value = 0;
5.  // оставив инструкцию вывода типа с explicit

6.  // можно разрешить
7.  Dummy dummy{ value, 1 };
8.  // и запретить
9.  Dummy dummy = { value, 1 };
10. // (конструктор гипо-класса стал explicit)

11. // сделав наоборот:
12. Dummy dummy = { value, 1 };
13. // конструктор класса - explicit, а у инструкции вывода explicit убрать
14. // процесс вывода типа Dummy<int> закончится успешно,
15. // но компиляция всё равно закончится ошибкой
```

Вывод типов шаблонных параметров

яснее ясного

1. `template <typename T, typename U> explicit Dummy(T&&, U&&) -> Dummy<T>;`
2. `// исходные explicit сущности`
3. `// получат в гипо-классе соответствующие explicit конструкторы`
4. `int value = 0;`
5. `// оставив инструкцию вывода типа с explicit`
6. `// можно разрешить`
7. `Dummy dummy{ value, 1 };`
8. `// и запретить`
9. `Dummy dummy = { value, 1 };`
10. `// (конструктор гипо-класса стал explicit)`
11. `// сделав наоборот:`
12. `Dummy dummy = { value, 1 };`
13. `// конструктор класса - explicit, а у инструкции вывода explicit убрать`
14. `// процесс вывода типа Dummy<int> закончится успешно,`
15. `// но компиляция всё равно закончится ошибкой`
16. Процесс вывода типа аргумента шаблона и процесс выбора конструктора
17. - это два несвязанных процесса.

Приятные мелочи

C++17: Встроенные переменные

обход правила одного определения (one definition rule) ([P0386R2](#))

```
1.  // C++17
2.  class SomeClass {
3.  public:
4.      // определение inline переменной в заголовочном файле
5.      static inline int myStaticVar{};
6.      // объявление и определение
7.      static constexpr int myStaticConst = 44;
8.  };

9.  // C++14
10. // компоновщик будет ругаться на отсутствие определений
11. // myStaticVar и myStaticConst
```

Вложенные пространства имён

НОВЫЙ СИНТАКСИС

```
1.  // C++17
2.  namespace CoolApp::Model::Instruments {
3.      class FakeGenerator {
4.          //...
5.      };
6.  }
```

```
7.  // C++14
8.  namespace CoolApp {
9.      namespace Model {
10.         namespace Instruments {
11.             class FakeGenerator {
12.                 // ...
13.             };
14.         }
15.     }
16. }
```

Выделение памяти с особым выравниванием

- В C++11 добавили функционал, позволяющий указать выравнивание для типа или для объекта в *стеке*

```
1.  #include <memory>
2.  #include <iostream>
3.  using namespace std;
4.  class alignas(32) SomeClass { char value; };

5.  int main() {
6.      auto some = make_unique<SomeClass>();
7.      bool isProperlyAligned = reinterpret_cast<size_t>(some.get()) % 32 == 0;
8.      cout << boolalpha << isProperlyAligned << "\n";
9.  };
```

- логично предположить, что код должен всегда выводить **true**

Выделение памяти с особым выравниванием

- В C++11 добавили функционал, позволяющий указать выравнивание для типа или для объекта в *стеке*

```
1.  #include <memory>
2.  #include <iostream>
3.  using namespace std;
4.  class alignas(32) SomeClass { char value; };

5.  int main() {
6.      auto some = make_unique<SomeClass>();
7.      bool isProperlyAligned = reinterpret_cast<size_t>(some.get()) % 32 == 0;
8.      cout << boolalpha << isProperlyAligned << "\n";
9.  };
```

- логично предположить, что код должен всегда выводить **true**, но компилятор не обязан выравнивать объекты, выделенные в куче, до C++17 ([P0035R4](#)),
- оператор **new** не имел перегрузки, в которую бы эта информация передавалась

Предикат препроцессора

проверить, доступен ли заголовочный файл для подключения ([P0061R1](#))

```
1.  #if __has_include(<optional>)
2.  #include <optional>
3.  #define have_optional 1
4.  #elif __has_include(<experimental/optional>)
5.  #include <experimental/optional>
6.  #define have_optional 1
7.  #define experimental_optional 1
8.  #else
9.  #define have_optional 0
10. #endif
```

C++17: Байт

`std::byte` введён как часть стандартной библиотеки, определён в `<cstdint>` ([P0298R3](#))

1. `enum class byte : unsigned char {};`

Инициализация в ветвлениях

можно определять прямо в условии оператора [else] if или switch

```
1.  // C++17
2.  mutex guard;
3.  if (lock_guard locker{ guard }; isContainerReady) {
4.      // ... работаем с контейнером
5.  }
```

```
6.  // C++14
7.  mutex guard;
8.  {
9.      lock_guard locker{ guard };
10.     if (isContainerReady) {
11.         // ... работаем с контейнером
12.     }
13. }
```

Инициализация в ветвлениях

```
1.  if (int x = 0; true) {
2.      x = 2; // y здесь не виден!
3.  }
4.  else if (int y = 3; y == 3) {
5.      cout << "y=" << y << ", x=" << x << "\n";
6.  } else {
7.      int z = x = y;
8.  }
9.  // x, y, z здесь не видны!
```

мотивирующий пример функционала из оригинального предложения ([P0305R1](#))

```
10. if ( auto keywords = { "if", "for", "while" };
11.     std::any_of(keywords.begin(), keywords.end(), [&s](const char* kw) {
12.         return s == kw;
13.     }) )
14. {
15.     ERROR("Token must not be a keyword");
16. }
```

constexpr lambda-функции

лямбды можно писать внутри constexpr выражений

```
1.  // C++17
2.  constexpr int Func(int x)
3.  {
4.      auto f = [x]() { return x * x; };
5.      return x + f();
6.  }

7.  int main()
8.  {
9.      constexpr int v = Func(10);
10.     static_assert(v == 110);

11.     constexpr auto squared = [](int x) { return x * x; };
12.     constexpr int s = squared(5);
13.     static_assert(s == 25);

14.     return 0;
15. }
```

*this в lambda-функции

захватывать члены класса по значению

```
1.  class SomeClass {
2.      int m_x = 0;
3.  public:
4.      void f() const { std::cout << m_x << '\n'; }
5.      void g() { m_x++; }

6.      // C++17
7.      void FuncNew() {
8.          // const copy *this
9.          auto lambda1 = [*this]() { f(); };
10.         // non-const copy *this
11.         auto lambda2 = [*this]() mutable { g(); };
12.         lambda1();
13.         lambda2();
14.     }
15. };
```

*this в lambda-функции

захватывать члены класса по значению

```
1.  class SomeClass {
2.      int m_x = 0;
3.  public:
4.      void f() const { std::cout << m_x << '\n'; }
5.      void g() { m_x++; }

6.      // C++14
7.      void Func() {
8.          // const copy *this
9.          auto lambda1 = [self = *this]() { self.f(); };
10.         // non-const copy *this
11.         auto lambda2 = [self = *this]() mutable { self.g(); };
12.         lambda1();
13.         lambda2();
14.     }
15. };
```

constexpr if

C++17: constexpr и привязки

- позволяет принимать решения на основании аргументов шаблона, а не является слепым инструментом подмены текста, как в макросах
- ***[else] if constexpr ((условие))***
- можно включать/выключать часть кода функции

C++17: constexpr и привязки

static if в C++

```
1.  template <typename T>
2.  void simpleExample(T val) {
3.      if constexpr (std::is_integral<T>::value)
4.          std::cout << "Integral passed.";
5.      else
6.          std::cout << "Non integral passed.";
7.      std::cout << "\n";
8.  }
```

C++17: constexpr и привязки

static if в C++

```
1.  template <typename T>
2.  void mixStaticWithDynamicIncorrect(T val) {
3.      if constexpr (std::is_integral<T>::value)
4.          std::cout << "Integral passed.";

5.      else if (val == std::string{ "clone" })
6.          std::cout << "Known string passed.";
7.      else if constexpr (std::is_same_v<T, std::string>)
8.          std::cout << "General string passed.";

9.      else
10.         std::cout << "Unknown type variable passed.";
11.         std::cout << "\n";
12. }

13. mixStaticWithDynamicIncorrect(1);
14. mixStaticWithDynamicIncorrect("clone"s);
15. mixStaticWithDynamicIncorrect("unique"s);
16. mixStaticWithDynamicIncorrect(1.0);
```

C++17: constexpr и привязки

static if в C++

```
1.  template <typename T>
2.  void mixStaticWithDynamicCorrect(T val) {
3.      if constexpr (std::is_integral<T>::value)
4.          std::cout << "Integral passed.";
5.      else if constexpr (std::is_same_v<T, std::string>) {
6.          if (val == std::string{ "clone" })
7.              std::cout << "Known string passed.";
8.          else
9.              std::cout << "General string passed.";
10.     }
11.     else
12.         std::cout << "Unknown type variable passed.";
13.     std::cout << "\n";
14. }

15. mixStaticWithDynamicIncorrect(1);
16. mixStaticWithDynamicIncorrect("clone"s);
17. mixStaticWithDynamicIncorrect("unique"s);
18. mixStaticWithDynamicIncorrect(1.0);
```

constexpr - немacroподстановка

не макросы, которые просто отключают один кусок текста

```
1.  template <typename T>
2.  void testStaticAssert(T val) {
3.      if constexpr (std::is_integral<T>::value)
4.          std::cout << "Integral passed.";
5.      else
6.          static_assert(false);
7.  }
```

constexpr - немacroподстановка

не макросы, которые просто отключают один кусок текста

```
1.  template <typename T>
2.  void testStaticAssert(T val) {
3.      if constexpr (std::is_integral<T>::value)
4.          std::cout << "Integral passed.";
5.      else
6.          //static_assert(false);
7.          static_assert(std::is_integral<T>::value);
8.  }
```

constexpr if - возвращаемый тип

возвращать разные типы из одной функции (естественно, не одновременно)

```
1.  template<typename T>
2.  auto returnHeadache(T val) {
3.      if constexpr (std::is_same_v<T, std::string>)
4.          return 0;
5.      else
6.          return std::string{ "str" };
7.      //return std::vector{ 1, 2, 3 }; // так нельзя!
8.  }
```

constexpr if

там, где раньше был SFINAE и enable_if

```
1.  #include <iostream>
2.  #include <type_traits>

3.  // C++14
4.  template <typename T>
5.  typename std::enable_if<std::is_pointer<T>::value, std::remove_pointer_t<T>>::type
6.  GetValue(T t) {
7.      return *t;
8.  }

9.  template <typename T>
10. typename std::enable_if<!std::is_pointer<T>::value, T>::type
11. GetValue(T t) {
12.     return t;
13. }

14. int main() {
15.     int v = 10;
16.     std::cout << GetValue(v) << '\n'; // 10
17.     std::cout << GetValue(&v) << '\n'; // 10
18.     return 0;
19. }
```

constexpr if

теперь достаточно if

```
1.  #include <iostream>
2.  #include <type_traits>

3.  // C++17
4.  template <typename T>
5.  auto GetValue(T t) {
6.      if constexpr (std::is_pointer<T>::value)
7.          return *t;
8.      else
9.          return t;
10. }

11.
12.
13.
14.

15. int main() {
16.     int v = 10;
17.     std::cout << GetValue(v) << '\n'; // 10
18.     std::cout << GetValue(&v) << '\n'; // 10
19.     return 0;
20. }
```

constexpr if

ещё пример SFINAE

```
1.  template<typename Container>
2.  std::enable_if_t<HasFunctionSort_v<Container>>
3.  sort(Container& container) {
4.      std::cout << "Calling member sort function\n";
5.      container.sort();
6.  }

7.  template<typename Container>
8.  std::enable_if_t<!HasFunctionSort_v<Container>>
9.  sort(Container& container) {
10.     std::cout << "Calling std::sort function\n";
11.     sort(std::begin(container), std::end(container));
12. }
```

constexpr if

и без него SFINAE

```
1.  // C++17
2.  template<typename Container>
3.  void sort(Container& container) {
4.      if constexpr (HasFunctionSort_v<Container>) {
5.          std::cout << "Calling member sort function\n";
6.          container.sort();
7.      } else {
8.          std::cout << "Calling std::sort function\n";
9.          sort(begin(container), end(container));
10.     }
11. }
```

constexpr if

при реализации класса `Function<R(Args...)>` за нами оставалась реализация функции

```
1.  template <typename R, typename... Args>
2.  R Function<R(Args...)>::operator()(Args... args) const override {
3.      return invokeImpl(functor, args...);
4.  }

5.  template <typename Functor, typename... Args,
6.            typename = std::enable_if_t
```

constexpr if

tag dispatching by instance

```
1.  template<typename Container>
2.  void sortImpl(Container& container, std::true_type) {
3.      std::cout << "Calling member sort function\n";
4.      container.sort();
5.  }

6.  template<typename Container>
7.  void sortImpl(Container& container, std::false_type) {
8.      std::cout << "Calling std::sort function\n";
9.      sort(begin(container), end(container));
10. }

11. template<typename Container>
12. void sort(Container& container) {
13.     sortImpl(container, HasFunctionSort<Container>{});
14. }
```

constexpr if

ограничения – нельзя создать такой код

```
1.  template <typename T>
2.  struct StaticIfMethod {
3.      template <typename U = T, typename = std::enable_if_t<std::is_integral_v<U>>>
4.      static void branch(std::string str) {
5.          std::cout << "String branch.\n";
6.      }
7.
8.      template<typename U = T, typename = std::enable_if_t<!std::is_integral_v<U>>>
9.      static void branch(int) {
10.         std::cout << "Integer branch.\n";
11.     };
};
```

Структурное связывание

Структурное связывание (structured bindings)

- Декомпозиция структур (классов) на части
- Облегчённый аналог **Javascript destructuring** или **C# deconstructing**

```
1. struct SimpleStruct {
2.     std::string first{ "first" };
3.     int second = 2;
4.     float third = 3.0;
5. };

6. int main() {
7.     // разберём её на кусочки
8.     auto [f, s, t] = SimpleStruct{};
9.     std::cout << "first: " << f
10.             << ", second: " << s
11.             << ", third: " << t << "\n";
12.     return 0;
13. }
```

Структурное связывание (structured bindings)

как это работает?

```
1.     auto tmp = SimpleStruct{};
2.     auto& f = tmp.first;
3.     auto& s = tmp.second;
4.     auto& t = tmp.third;
```

Структурное связывание (structured bindings)

как это работает?

```
1.     auto tmp = SimpleStruct{};
2.     auto& f = tmp.first;
3.     auto& s = tmp.second;
4.     auto& t = tmp.third;
```

массивы

```
1.     int array[] = { 1, 3, 5, 7, 9 };
2.     auto[a1, a2, a3, a4, a5] = array;
3.     std::cout << "Third element before: "
4.               << a3 << "(" << array[2] << ")\n";
5.     a3 = 4;
6.     std::cout << "Third element after: "
7.               << a3 << "(" << array[2] << ")\n";
```

Структурное связывание (structured bindings)

АТД

```
1.  class Person {
2.      std::string m_FirstName;
3.      std::string m_LastName;
4.      size_t m_Id;
5.  public:
6.      Person(const std::string& first, const std::string& last, size_t id) :
7.          m_FirstName{ first },
8.          m_LastName{ last },
9.          m_Id{ id }
10.     {}
11.     const std::string& firstName() const noexcept { return m_FirstName; }
12.     const std::string& lastName() const noexcept { return m_LastName; }
13.     size_t id() const noexcept { return m_Id; }
14. };

15. int main() {
16.     Person einstein{ "Albert", "Einstein", 997 };
17.     auto[id, first, last] = einstein; // нельзя получить доступ к закрытым полям!

18.     return 0;
19. }
```

Структурное связывание (structured bindings)

АТД

```
1. namespace std {  
2.     // мы действительно добавляем новый тип в пространство имён std!  
  
3.     // число элементов, которые будут представлены классом  
4.     template <> struct tuple_size<Person> { constexpr static size_t value = 3; };  
5. } // std
```

Структурное связывание (structured bindings)

АТД

```
1. namespace std {  
2.     // мы действительно добавляем новый тип в пространство имён std!  
  
3.     // число элементов, которые будут представлены классом  
4.     template <> struct tuple_size<Person> { constexpr static size_t value = 3; };  
5.     // типы этих полей  
6.     template <> struct tuple_element<0, Person> { using type = size_t; };  
7.     template <> struct tuple_element<1, Person> { using type = std::string; };  
8.     template <> struct tuple_element<2, Person> { using type = std::string; };  
9. } // std
```

Структурное связывание (structured bindings)

АТД

```
1. namespace std {
2.     // мы действительно добавляем новый тип в пространство имён std!

3.     // число элементов, которые будут представлены классом
4.     template <> struct tuple_size<Person> { constexpr static size_t value = 3; };
5.     // типы этих полей
6.     template <> struct tuple_element<0, Person> { using type = size_t; };
7.     template <> struct tuple_element<1, Person> { using type = std::string; };
8.     template <> struct tuple_element<2, Person> { using type = std::string; };

9.     // на каждое поле нужно добавить функцию get
10.    template <size_t Position> auto get(const Person&) = delete;
11.    template <> auto get<0>(const Person& person) { return person.id(); }
12.    template <> auto get<1>(const Person& person) { return person.firstName(); }
13.    template <> auto get<2>(const Person& person) { return person.lastName(); }
14. } // std
```

Структурное связывание (structured bindings)

АТД

```
1. namespace std {
2.     // мы действительно добавляем новый тип в пространство имён std!

3.     // число элементов, которые будут представлены классом
4.     template <> struct tuple_size<Person> { constexpr static size_t value = 3; };
5.     // типы этих полей
6.     template <> struct tuple_element<0, Person> { using type = size_t; };
7.     template <> struct tuple_element<1, Person> { using type = std::string; };
8.     template <> struct tuple_element<2, Person> { using type = std::string; };

9.     // на каждое поле нужно добавить функцию get
10.    template <size_t Position> auto get(const Person& person) {
11.        if constexpr (Position == 0)
12.            return person.id();
13.        else if constexpr (Position == 1)
14.            return person.firstName();
15.        else if constexpr (Position == 2)
16.            return person.lastName();
17.    }
18. } // std
```

Структурное связывание (structured bindings)

АТД

```
1. namespace std {
2.     // мы действительно добавляем новый тип в пространство имён std!

3.     // число элементов, которые будут представлены классом
4.     template <> struct tuple_size<Person> { constexpr static size_t value = 3; };
5.     // типы этих полей
6.     template <> struct tuple_element<0, Person> { using type = size_t; };
7.     template <> struct tuple_element<1, Person> { using type = std::string; };
8.     template <> struct tuple_element<2, Person> { using type = std::string; };

9.     // на каждое поле нужно добавить функцию get
10.    template <size_t Position> decltype(auto) get(const Person& person) {
11.        if constexpr (Position == 0)
12.            return person.id();
13.        else if constexpr (Position == 1)
14.            return person.firstName();
15.        else if constexpr (Position == 2)
16.            return person.lastName();
17.    }
18. } // std
```

Структурное связывание (structured bindings)

АТД

```
1.  class Person {
2.      std::string m_FirstName;
3.      std::string m_LastName;
4.      size_t m_Id;
5.  public:
6.      Person(const std::string& first, const std::string& last, size_t id) :
7.          m_FirstName{ first },
8.          m_LastName{ last },
9.          m_Id{ id }
10.     {}
11.     const std::string& firstName() const noexcept { return m_FirstName; }
12.     const std::string& lastName() const noexcept { return m_LastName; }
13.     size_t id() const noexcept { return m_Id; }
14. };

15. int main() {
16.     Person einstein{ "Albert", "Einstein", 997 };
17.     const auto& [id, first, last] = einstein;
18.     id = 998; // нельзя
19.     return 0;
20. }
```

Структурное связывание (structured bindings)

применение

```
1.  using Vector3d = std::tuple<int, int, int>;
2.  Vector3d getVector() { return std::make_tuple(0, 1, 2); }
3.  int main() {
4.      int x = 0, y = 0, z = 0;
5.      std::tie(x, y, z) = getVector();
6.
7.      auto [x, y, z] = getVector();
8.      std::tie(std::ignore, std::ignore, z) = getVector(); // можно
9.      auto [_, _, z] = getVector(); // а так – нет, когда-нибудь?
10.
11.     return 0;
12. };
```

Структурное связывание (structured bindings)

применение

```
1.  using Vector3d = std::tuple<int, int, int>;
2.  Vector3d getVector() { return std::make_tuple(0, 1, 2); }
3.  int main() {
4.      int x = 0, y = 0, z = 0;
5.      std::tie(x, y, z) = getVector();
6.
7.      auto [x, y, z] = getVector();
8.
9.      std::tie(std::ignore, std::ignore, z) = getVector(); // можно
10.     auto [_, _, z] = getVector(); // а так - нет, когда-нибудь?
11.
12.     std::map<std::string, size_t> albums = {
13.         {"Coven", 1991},
14.         {"Fool", 1997}
15.     };
16.     auto [position, inserted] = albums.emplace("Outcast", 2005);
17.     for (const auto& [name, year] : albums)
18.         std::cout << name << ": " << year << "\n";
19.     return 0;
20. }
```

Порядок выполнения инструкций

- `int i = 0;`
- `i = i++ + i++;`
- Спрашивают на собеседованиях, *декларируемая цель*: знаком ли порядок исполнения выражений в C++.
- Корректно ли такое спрашивать?
- Можно ознакомиться с мыслями Реймонда Чена на эту тему: «*Do people write insane code with multiple overlapping side effects with a straight face?*»
- <https://blogs.msdn.microsoft.com/oldnewthing/20170719-00/?p=96645>

Порядок выполнения

Есть менее очевидные случаи...

- В последней книге Бьёрна Страуструпа
- «The C++ Programming Language 4th edition»
- <http://scrutator.me/admin/app/editor/CppProgrammingLanguage4EdReview>

```
1. void f2() {
2.     std::string s = "but I have heard it works even if you don't believe in it";
3.     s.replace(0, 4, "")
4.         .replace(s.find("even"), 4, "only")
5.         .replace(s.find(" don't"), 6, "");
6.     assert(s == "I have heard it works only if you believe in it");
7. }

8. // не нужно напихивать кучу кода в одно выражение,
9. // в котором происходит масса всяких разных вещей.
10. void f() {
11.     std::string s = "but I have heard it works even if you don't believe in it";
12.     s.replace(0, 4, "");
13.     s.replace(s.find("even"), 4, "only");
14.     s.replace(s.find(" don't"), 6, "");
15.     assert(s == "I have heard it works only if you believe in it");
16. }
17. // с порядком исполнения подвыражений выражений в C++ что-то не так (P0145R3)
```

Старый порядок

- В C++ порядок исполнения подвыражений в выражениях не определяется стандартом и остаётся на откуп компилятору.
- Некоторый порядок есть, но тонкостей достаточно много
- 2 подвыражения одного большого выражения исполняются независимо друг от друга в *неопределённом* порядке (исключение: оператор запятая «,»)
- **`i = i++ + i++;`**
- C++14 гарантирует (***expr.ass***), что оба выражения `i++` будут вычислены до того, как будет вычислена их сумма, а также то, что выражение `i` будет вычислено до того, как ему будет присвоен результат суммы
- Компилятор может выбрать несколько путей вычисления полного выражения: он не ограничен в том, когда эффект от `++` должен быть применён к `i`
- После вычисления этого выражения `i` может остаться = 0

Старый порядок

- *неопределённое поведение*
(UB - undefined behavior)
- согласно стандарту (*intro.execution/p15*) выражение является **UB** уже потому, что
- «в одном выражении присутствуют два подвыражения, которые модифицируют один и тот же скалярный объект, и при этом порядок изменений не определён»
- согласно текущим правилам стандарта нет другого выхода, как развести руками
 - *std::map<int, size_t> dictionary;*
 - *dictionary[0] = dictionary.size();*

Старый порядок

- `std::cout << first << second;`
- В зависимости от того, чем являются `first` и `second`, можно получить
 - `std::cout.operator<<(first).operator<<(second);`
 - `operator<<(operator<<(std::cout, first), second);`
- 3 выражения и 2 оператора `<<` - никаких функций!

```
1.  template <typename T>
2.  bool cleverFun(T& value)
3.  {
4.      return (cout << "first\n", value++) &&
5.              (cout << "second\n", value++);
6.  }
```

Старый порядок

- В C++14 гарантии для операторов и их операндов отличаются в зависимости от того, чем являются операнды.
- Для интегральных типов все гарантии операторов работают так, как они описаны для них в стандарте.
- Для переопределённых операторов работают правила вызовом функций.
- **Никакие гарантии операторов из стандарта на переопределённые операторы не действуют.**

C++17: Новый порядок

- упорядочивание выполнения
 - постфиксных операторов и
 - операторов побитового сдвига (слева направо),
 - операторов присвоения (справа налево)
- «*Выполнение*» - выражение является вычисленным, т.е. его результат возвращён, и все ассоциированные побочные эффекты зафиксированы (committed)
 - **a.b**
 - **a->b**
 - **a->*b**
 - **a(b1, b2, b3)**
 - **b @= a**
 - **a[b]**
 - **a << b**
 - **a >> b**

C++17: Новый порядок

- правила не коснулись порядка выполнения **аргументов функции** относительно друг друга:
 - по-прежнему могут быть выполнены в любом порядке
 - их выполнение не может пересекаться (interleave) между собой
 - они упорядочены относительно друг друга, но порядок не регламентирован

C++17: Новый порядок

- `i = i++;`
- `f(++i, ++i)`
- `f(i++, i++)`
- `array[i++] = i++`
- `i << i++`
- `cout << i++ << i++`

C++17: Новый порядок

- `i = i++;`
- `f(++i, ++i)`
- `f(i++, i++)`
- `array[i++] = i++`
- `i << i++`
- `cout << i++ << i++`

Остались некорректны

- `i = i++ + i++`
- `i = ++i * i++`

C++17: Новый порядок

- `i = i++;`
- `f(++i, ++i)`
- `f(i++, i++)`
- `array[i++] = i++`
- `i << i++`
- `cout << i++ << i++`

Остались некорректны

- `i = i++ + i++`
- `i = ++i * i++`
- никаких правил, регулирующих порядок выполнения подвыражений арифметических операторов, не добавили

"C++17: Новый порядок

- «Логика переписывания»
- Берём
 - `cout << i++ << i++;`

"C++17: Новый порядок

- «Логика переписывания»
- Берём
 - `cout << i++ << i++;`
- Мысленно представляем
 - `cout.operator<<(i++).operator<<(i++);`

"C++17: Новый порядок

- «Логика переписывания»
- Берём
 - `cout << i++ << i++;`
- Мысленно представляем
 - `cout.operator<<(i++).operator<<(i++);`
- Пример действительно «переписывается» компилятором, но
 - Порядок исполнения выстраивается до переписывания!
 - Согласно новым правилам, *«перегруженные операторы подчиняются правилам исполнения для встроенных операторов»* (в части порядка подвыражений)
 - Исходя из правила, что *«левый операнд оператора вычисляется до правого»*, нет **UB**

"C++17: Новый порядок

- это нововведение породило разницу между явным и неявным вызовом перегруженного оператора
 - `int result = SomeClass{i = 1} << (i = 2);`
 - `result = operator<<(SomeClass{i = 1}, i = 2);`
- Теперь интерфейс вполне поощряет использование цепочки вызовов

Пропуск копирования (copy elision)

сколько раз на экране появится фраза «Copy ctor called»?

```
1.  #include <iostream>
2.  using namespace std;

3.  class SomeClass {
4.  public:
5.      SomeClass() = default;
6.      SomeClass(const SomeClass&) {
7.          cout << "Copy ctor called\n";
8.      }
9.  };
10. SomeClass meReturn() {
11.     return SomeClass{};
12. }

13. int main() {
14.     auto some = meReturn();
15. };
```

Пропуск копирования (copy elision)

сколько раз на экране появится фраза «Copy ctor called»?

```
1.  #include <iostream>
2.  using namespace std;

3.  class SomeClass {
4.  public:
5.      SomeClass() = default;
6.      SomeClass(const SomeClass&) {
7.          cout << "Copy ctor called\n";
8.      }
9.  };
10. SomeClass meReturn() {
11.     SomeClass some{};
12.     return some;
13. }

14. int main() {
15.     auto some = meReturn();
16. };
```

Пропуск копирования (copy elision)

сколько раз на экране появится фраза «Copy ctor called»?

```
1.  #include <iostream>
2.  using namespace std;

3.  class SomeClass {
4.  public:
5.      SomeClass() = default;
6.      SomeClass(const SomeClass&) {
7.          cout << "Copy ctor called\n";
8.      }
9.  };
10. SomeClass meReturn() {
11.     SomeClass some{};
12.     if (false) return SomeClass{};
13.     return some;
14. }
15. int main() {
16.     auto some = meReturn();
17. };
```

Пропуск копирования (copy elision)

- Согласно стандарту C++, в некоторых случаях компилятор может не выполнять копирования объекта
- Полный список **[class.copy/p31]** довольно короткий
 - Функция возвращает временный безымянный объект - компилятор имеет право опустить оба копирования и создать объект напрямую в **some**
оптимизация возвращаемого значения (return value optimization)
 - компилятор имеет право избавиться от копирования именованного объекта на стеке, который возвращается из функции
оптимизация именованного возвращаемого значения (named return value optimization).
- МОЖНО заставить создавать все копии
 - **gcc/clang** с ключом **-fno-elide-constructors**
 - MSVC ключей отключения этого поведения не имеет

Новые атрибуты C++11

- `[[noreturn]]`
- `[[carries_dependency]]`

Новые атрибуты C++14

- `[[noreturn]]`
- `[[carries_dependency]]`
- `[[deprecated]]`

Новые атрибуты C++17

- `[[noreturn]]`
- `[[carries_dependency]]`
- `[[deprecated]]`
- `[[maybe_unused]]`
- `[[nodiscard]]`
- `[[fallthrough]]`

[[noreturn]]

функция не возвращает управление привычным путём

```
1.  #include <cstdlib>

2.  [[noreturn]] void hard_stop() {
3.      std::exit(1);
4.  }

5.  [[noreturn]] void throw_anyway() {
6.      throw "anyway";
7.  }
```

[[carries_dependency]]

позволяет переносить зависимости по данным между потоками в вызовы функций

```
1.  #include <atomic>
2.  #include <iostream>
3.  using namespace std;

4.  void print(int * val) {
5.      cout << *val << endl;
6.  }
7.  void print2(int *[[carries_dependency]] val) {
8.      cout << *val << endl;
9.  }

10. void main()
11. {
12.     atomic<int*> p;
13.     int* local = p.load(memory_order_consume);
14.     if (local)
15.         cout << *local << endl; // #1
16.     if (local) print(local);      // #2
17.     if (local) print2(local);    // #3
18. }
```

[[deprecated]]

принципиально новый синтаксис атрибутов

```
1.  [[deprecated]]
2.  void foo() {}
3.  [[deprecated("use baz instead")]]
4.  void bar() {}

5.  int main() {
6.      foo(); // warning: 'foo' is deprecated.
7.           // note: 'foo' has been explicitly marked deprecated here
8.      bar(); // warning: 'bar' is deprecated: use baz instead.
9.           // note: 'bar' has been explicitly marked deprecated here
10.     return 0;
11. }
```

[[deprecated]]

принципиально новый синтаксис атрибутов

1. `[[deprecated]]`
2. `int x;`
3. `int y[[deprecated]], z; // помечаем только одну переменную в списке определяемых`
4. `int triple([[deprecated]] int x);`
5. `template <typename T> class templ;`
6. `template <>`
7. `class [[deprecated]] templ<int> {}; // помечаем специализацию шаблона`
8. `struct [[deprecated]] A {`
9. `[[deprecated]] int member;`
10. `};`
11. `[[deprecated]] typedef int type;`

Новые атрибуты C++17

- Атрибуты теперь можно применять к **пространствам имён и членам перечислений** ([N4266](#))
- Использовать **using** в атрибутах, чтобы избавиться от повторения пространства имён для каждого атрибута в списке ([P0028R4](#))
- **Неизвестные атрибуты** больше не дают ошибку компиляции, а просто игнорируются ([P0283R2](#))

[[fallthrough]]

оператор break внутри блока case отсутствует намеренно

```
1.  switch (i) {
2.      case 10:
3.          f1();
4.          break;
5.      case 20:
6.          f2();
7.          break;
8.      case 30:
9.          f3();
10.         break;
11.     case 40:
12.         f4();
13.         [[fallthrough]]; // предупреждение будет подавлено
14.     case 50:
15.         f5();
16. }
```

[[nodiscard]]

- **исключений vs коды возврата**
- выброшенное **исключение проигнорировать *случайно* нельзя**, а возвращённое значение функции — запросто
- скрестив исключения с кодами возврата («[Systematic Error Handling in C++](#)»), Андрей Александреску получил **Expected**,
- что привело к появлению **std::expected**

[[nodiscard]]

применение

```
1. class [[nodiscard]] Expected {
2.     // ...
3. };

4. [[nodiscard]] int importantFunction() {
5.     return 24;
6. }

7. Expected expectedFunction() {
8.     return {};
9. }

10. int main() {
11.     importantFunction(); // warning: возвращённый из функции объект проигнорирован
12.     expectedFunction(); // warning: возвращённый из функции объект проигнорирован
13.     return 0;
14. };
```

[[nodiscard]]

особенность

```
1.  #include <optional>
2.  #include <memory>
3.  using namespace std;

4.  struct IStreamReader {
5.      [[nodiscard]] virtual optional<char> read() = 0;
6.  };
7.  struct FakeStreamReader : public IStreamReader {
8.      optional<char> read() override {
9.          return '\0';
10.     }
11. };

12. int main() {
13.     unique_ptr<IStreamReader> smartStream = make_unique<FakeStreamReader>();
14.     FakeStreamReader fakeStream;
15.     smartStream->read();    // #1.
16.     fakeStream.read();     // #2.
17. }
```

[[nodiscard]]

возвращаемое значение функции должно быть обязательно использовано при вызове

```
1.  [[nodiscard]] int Sum(int a, int b) {
2.      return a + b;
3.  }
4.  int main() {
5.      Sum(5, 6); // будет выдано предупреждение компилятора/анализатора
6.      return 0;
7.  }

8.  struct [[nodiscard]] NoDiscardType {
9.      char a;
10.     int b;
11. };
12. NoDiscardType Func() {
13.     return { 'a', 42 };
14. }

15. int main() {
16.     Func(); // предупреждение компилятора/анализатора
17.     return 0;
18. }
```

[[maybe_unused]]

Введение имени, которое нигде не используется – довольно распространённое явление:

- **__attribute__((unused))** в **gcc**,
- **Q_UNUSED** в **Qt**,
- **ignore_unused** в **boost** и т.д.

Зачем вообще вводить именованный объект, который нигде не используется?

[[maybe_unused]]

подавить предупреждения компилятора/анализатора о неиспользуемой переменной, параметре функции, статической функции и пр.

```
1. void processingFunction(int important, [[maybe_unused]] int threshold)
2. {
3.     assert(important < threshold);
4.     // Используем important
5.     //...
6.     [[maybe_unused]] TraceLogger logger{ __FUNCTION__ };
7.     (void)logger;
8.     //...
9. }
```

C++17: обновления библиотеки

C++17: обновления библиотеки

- <https://isocpp.org/files/papers/p0636r0.html>
- **pmr::memory_resource** — полиморфные и type-erased аллокаторы
- Новые алгоритмы **for_each_n**, **reduce**, **transform_reduce**, **exclusive_scan**, **inclusive_scan**, **transform_exclusive_scan**, **transform_inclusive_scan**
- математические функции — из стандарта **ISO/IEC 29124:2010**, плюс ещё **gcd** и **lcm**, **hypot**(x, y, z), **clamp**(x, low, high)
- Шестнадцатеричное представление чисел с плавающей точкой **0x3.ABCp-10**, **0x1.0p-126**
- **std::filesystem**, **std::any**, **std::optional**
- функции **std::size**, **std::empty**, **std::data**, **std::begin**, **std::end**
- **std::string::data()** возвращает неконстантный **char***

C++17: обновления библиотеки

- <https://isocpp.org/files/papers/p0636r0.html>
- **std::shared_ptr** для массивов, получение слабого указателя **std::shared_ptr::weak_type** без уничтожение имеющегося
- **std::result_of** → **std::invoke_result**
- **std::invoke**, **std::is_invocable**, **std::is_invocable_r**
- **std::void_t**
- **std::scoped_lock** – версия **std::lock_guard**, которая работает сразу со множеством мьютексов
- Функции, позволяют платформам документировать размер линии кэша процессора для программ
- Функторы поиска подстроки в строке, реализуют алгоритмы поиска **Boyer-Moore** и **Boyer-Moore-Horspool**
- **constexpr** для итераторов и функций-помощников **std::array**

C++17: обновления библиотеки

- <https://isocpp.org/files/papers/p0636r0.html>
- **std::to_chars/std::from_chars** — методы быстрых преобразований чисел в строки и наоборот, без использования **<locale>**
- **std::has_unique_object_representations<T>** — узнавать есть ли у типа единственное бинарное представление (для **memcmp**)
- **std::is_aggregate** — есть ли инициализация с **initialize_list**
- Новый псевдоним **template <bool B> using std::bool_constant = integral_constant<bool, B>**
- Traits for SFINAE-friendly **std::swap**
- **std::as_const(lvalue)**, возвращает версию с модификатором **const** (не связана с **rvalues**)
- **std::not_fn** корректно работает для **operator() const&**, **operator() &&** и пр.

C++17: обновления библиотеки

emplace_back контейнеров возвращает ссылку на созданный элемент

```
1.  void main()
2.  {
3.      std::vector<int> my_vector;
4.      // C++11
5.      my_vector.emplace_back();
6.      my_vector.back().do_something();

7.      // C++17
8.      my_vector.emplace_back().do_something();
9.  }
```

C++17: многопоточные алгоритмы

практически все алгоритмы STL продублированы с Execution Policy

```
1.  #include <vector>
2.  #include <algorithm>
3.  #include <execution>

4.  void function_fills_vector(std::vector<int>& v);

5.  void main()
6.  {
7.      std::vector<int> v;
8.      v.reserve(100500 * 1024);
9.      function_fills_vector(v);

10.     // многопоточная сортировка данных
11.     std::sort(std::execution::par, v.begin(), v.end());

12.     std::sort(std::execution::par, v.begin(), v.end(), [](auto left, auto right) {
13.         if (left == right) // вызовет std::terminate()
14.             throw std::logic_error("Equal values are not expected");
15.         return left < right;
16.     });
17. }
```

Передача внутренних структур хранения контейнеров наружу

потокбезопасная очередь с приоритетом

```
1.  #include <mutex>
2.  #include <set>
3.  template <typename T> class TaskQueue {
4.      using value_type = T;
5.      std::mutex values_mutex_;
6.      std::multiset<value_type> values_;
7.      std::condition_variable cond_;
8.
9.      // C++11
10.     void push(std::multiset<value_type>&& items) {
11.         std::unique_lock<std::mutex> lock(values_mutex_);
12.         for (auto&& val : items)
13.             values_.insert(val); // аллоцирует память, может кидать исключения
14.         cond_.notify_one();
15.     }
16.     value_type pop() {
17.         std::unique_lock<std::mutex> lock(values_mutex_);
18.         while (values_.empty())
19.             cond_.wait(lock);
20.         value_type ret = *values_.begin(); // аллоцирует память, может кидать исключения
21.         values_.erase(values_.begin()); // деаллоцирует память
22.         return ret;
23.     };
};
```

Передача внутренних структур хранения контейнеров наружу

потокбезопасная очередь с приоритетом

```
1.  #include <mutex>
2.  #include <set>
3.  template <typename T> class PriorityQueue {
4.      using value_type = T;
5.      std::mutex values_mutex_;
6.      std::multiset<value_type> values_;
7.      std::condition_variable cond_;
8.
9.      // C++17
10.     void push(std::multiset<value_type>&& items) {
11.         std::unique_lock<std::mutex> lock(values_mutex_);
12.         values_.merge(std::move(items)); // не аллоцирует память, не кидает исключения
13.         cond_.notify_one();
14.     }
15.     value_type pop() {
16.         std::unique_lock<std::mutex> lock(values_mutex_);
17.         while (values_.empty())
18.             cond_.wait(lock);
19.         auto node = values_.extract(values_.begin()); // извлекаем значение вершины
20.         lock.unlock();                                // не аллоцирует память и не бросает исключений
21.         return std::move(node.value());
22.     };
23. }
```

boost::string_ref → std::string_view

не владеет строкой, хранит указатель на её начало и размер

```
1.  // C++11
2.  #include <string>

3.  void get_vendor_from_id(const std::string &id) {
4.      // аллоцирует память, если передан массив символов вместо std::string
5.      std::cout << id.substr(0, id.find_last_of(':')); // аллоцирует память
6.  }

7.  // TODO: void get_vendor_from_id(const char*);

8.  // C++17
9.  #include <string_view>    // std::basic_string_view или std::string_view

10. void get_vendor_from_id(std::string_view id) {
11.     // не аллоцирует память, для `const char*`, `char*`, `const std::string&` и пр.
12.     std::cout << id.substr(0, id.find_last_of(':')); // не аллоцирует память
13. }
```

boost::variant → std::variant

- убраны все известные недочёты **boost::variant**
- никогда не аллоцирует память для собственных нужд
- методы **constexpr**, чтобы использовать в **constexpr** выражениях
- умеет делать **emplace**
- обращаться к значению можно по индексу или по типу
- не нуждается в **boost::static_visitor**
- не умеет рекурсивно держать в себе себя

убран функционал

```
boost::make_recursive_variant<int, std::vector<boost::recursive_variant>>>::type);
```

boost::variant → std::variant

int – 4 байта

double – 8 байт

std::string – 32 байта

std::variant<int, double, std::string> – 36 байт:

- 4 байта: индекс хранимого типа
- 32 байта: размер наибольшего типа (std::string)

std::variant

переменная, которая знает, какой именно тип она хранит

1. `std::variant<int, char, double> v;`
2. `v = 5;`
3. `std::cout << std::get<int>(v);`
4. `std::cout << std::get<double>(v);`
5. `std::cout << std::get<float>(v);`
6. `auto p = std::get_if<char>(&v);`

std::variant

переменная, которая знает, какой именно тип она хранит

```
1.  std::variant<int, char, double> v;  
2.  v = 5;  
  
3.  std::cout << std::get<int>(v);    // ok  
4.  std::cout << std::get<double>(v); // исключение std::bad_variant_access  
5.  std::cout << std::get<float>(v);  // compile error  
  
6.  auto p = std::get_if<char>(&v);  // nullptr
```

std::variant

использование шаблона проектирования Посетитель (Visitor)

```
1.  #include <boost/variant.hpp>
2.  #include <type_traits>
3.  #include <iostream>
4.  using namespace std;

5.  using Variant = boost::variant<int, unsigned char, std::string>;

6.  template <typename V>
7.  struct VisitorEqV : boost::static_visitor<bool> {
8.      const V value;
9.      VisitorEqV(V v) : value(v) {}
10.     template <typename T>
11.     bool operator()(T v) const { return (value == v); }
12.     bool operator()(const string& s) const { return (value == stoi(s)); }
13. };

14. void doVariant(const Variant &var) {
15.     const int value = 42;
16.     cout << boolalpha << boost::apply_visitor(VisitorEqV<int>{value}, var) << endl;
17. }
```

std::variant

использование шаблона проектирования Посетитель (Visitor)

```
1.  void main() {
2.      try {
3.          Variant var;
4.          boost::get<int>(var) = 42;
5.          doVariant(var);
6.          var = 33;
7.          doVariant(var);
8.          var = "33";
9.          doVariant(var);
10.         var = 'c';
11.         doVariant(var);
12.         var = "hello!";
13.         doVariant(var);
14.     }
15.     catch (std::exception &e) { cout << e.what() << endl; }
16. }
```

std::variant

использование шаблона проектирования Посетитель (Visitor)

```
1.  #include <boost/functional/overloaded_function.hpp>
2.  #include <boost/functional.hpp>
3.  #include <boost/bind.hpp>

4.  void main() {
5.      try {
6.          boost::variant<int, std::string> v;
7.          v = 5;
8.          int z = 7;

9.          auto overloaded = boost::make_overloaded_function(
10.             [z](const int i) -> void { cout << i << z << endl; },
11.             [z](const std::string& str) -> void { cout << str << z << endl; });
12.          auto binded = boost::bind<void>(overloaded, boost::placeholders::_1);
13.
14.          boost::apply_visitor(binded, v);
15.          v = "arrrrr";
16.          boost::apply_visitor(binded, v);
17.      }
18.      catch (std::exception &e) { cout << e.what() << endl; }
19.  }
```

Упражнение:

- Value or Exception
- написать 2 реализации **TValueOnError**
 - используя `std::variant`
 - и без него

Упражнение:

интерфейс использования ValueOrException:

```
1.  template <typename T> class TValueOrError { /* ??? */ };

2.  TValueOrError<int> getValue(char *str) {
3.      try {
4.          int param;
5.          if (str == nullptr || (param = std::stoi(str)) == 0)
6.              throw std::logic_error("param can't be zero");
7.          else
8.              return param / 2;
9.      } catch (...) { return std::current_exception(); }
10. }

11. int main(int argc, char **argv) {
12.     try {
13.         TValueOrError<int> result = getValue(argv[1]);
14.         std::cout << result.isError() << " "
15.                 << result.isValid() << std::endl;
16.         auto r = result.getValueOrThrow();
17.         // here can use it ...
18.         std::cout << "result = " << r << std::endl;
19.     } catch (std::logic_error& e) {
20.         std::cout << "error: " << e.what() << std::endl;
21.     }
22. }
```

СПАСИБО ЗА ВНИМАНИЕ!