Лекция 6.
New C++ features

CS Club, Novosibirsk, 2019

Часть 1. Небольшие приятные возможности

Alias templates (C++11)

- Появилась возможность делать typedef на шаблонный тип

```
1.  typedef
2.      std::map<std::string, size_t>
3.      words_counter_t;
4.
5.  template<class type>
6.  using obj_counter = std::map<type, size_t>;
7.
8.  using ivec = std::vector<int>;
9.
10. //..
11. obj_counter<int> digit_counter;
```

Range-based for (C++11)

- Намного компактнее итерирование по контейнеру, если не нужен индекс

```
    list<string> strings;

1.  for (list<string>::iterator it = strings.begin(); it !=
    strings.end(); ++it)
    {/*.*/}

2.  for (auto it = strings.begin(); it != strings.end(); ++it)
    {/*.*/}

3.  for (string& str: strings) {/*...*/}

4.  for (auto& str: strings) {/*...*/}
```

initializer list (C++11)

- копируется только по ссылке
- создается только синтаксисом через {}

```
1.  struct compl
2.  {
3.      double real;
4.      double img;
5.  };
6.  
7.  compl c = {3, 4};
8.  int a [10] = {1, 2, 3, 5, 7, 11};
9.  
10. vector<int> vi = {1, 2, 4, 8, 16, 32};
11. map<int, string> mis {{1, "one"}, {2, "two"}, {3, "three"}};`
12. 
13. struct seq
14. {
15.     seq(std::initializer_list<int> args)
16.     { assign(args.begin(), args.end()); }
17. };
18. 
19. // Uniform initialization
20. // one int equal to 4, or 4 ints with zero value?
    vector vi{4};
```

new string literals (C++11)

- Позволяют задавать строку в Unicode кодировке.
- Можно задать строку без экранирования символов.

```
1. u8"This is how to write in utf-8 & char \u2018"
2. u"This is how to write in utf-16 & char \u2018"
3. U"This is how to write in utf-32 & char \U00002018"
4.
5. R"(raw string "without" special \ symbols)"
6. u8R"delimiter(raw string "without" special \
7. symbols)delimiter";
```

- Не забывайте про библиотеки для l10n, например gettext

Delegating Constructors(C++11)

- Дефолтные значения уходят в реализацию.
- Объект считается созданным уже после первого конструктора.

```
1.  struct string
2.  {
3.      explicit string(const char* str) {/*...*/}
4.      string(string const& other)
5.          : string(other.c_str()){/*...*/}
6.  };
7.  
8.  struct def
9.  {
10.     def(size_t number) {/*...*/}
11.     def() : def(238)   {/*...*/}
12. };
```

Non-static data member initializers(C++11)

- Полю будет присвоено указанное значение, если его не перекрывает конструктор.

```
1.  class some
2.  {
3.  public:
4.      some() {}
5.      explicit some(int new_value) : value(new_value) {}
6.  
7.  private:
8.      int value    = 15;
9.      string name = "Stefan";
10. };
```

Часть 2. Большие возможности

auto / decltype

auto (C++11)

- Значительно упрощают указание типов.
- Выводит тип по тем же правилам, что и вывод шаблонного аргумента функции: отбрасывает top level ссылку и следующий за ней top level const и volatile спецификаторы (если есть).
- Можно написать код, который раньше был бы невозможен

auto (C++11)

```
1.  auto x = 5;
2.  auto it = vec.begin();
3.  auto& value = my_map[key];
4.  
5.  // trailing return type
6.  template<class T, class A>
7.  auto vector<T, A>::begin() -> iterator {/*...*/}
8.  
9.  // much easier iterator stuff
10. for (std::map<string, double>::const_iterator it = ...)
11. for (auto it = ...)
12. 
13. // impossible before
14. template<typename T, typename S>
15. void foo(T lhs, S rhs) {
16.         auto prod = lhs * rhs;
17.         //...
18. }
```

auto (C++14)

- Auto вывод возвращаемого параметра
  - Все return-выражения должны совпадать по типу (с оговорками auto)
  - Требуется определение до использования
  - Возможна рекурсия
- Generic lambdas

```
1.  template<class range_t>
2.  auto inversed (range_t range)
3.  {
4.      if (range.empty())
5.          return range;
6.  
7.      // inverse it somehow...
8.      return range;
9.  }
10. //...
11. auto lambda = [](auto x, auto y) { return x + y; };
12. 
```

decltype(C++11)

- Позволяет вывести точный тип выражения

```
1. template<typename T, typename S>
2. void foo(T lhs, S rhs) {
3.         typedef decltype(lhs * rhs) product_type;
4.         //...
   }
```

- Более гибкие правила, если выражение
  - переменная без скобок, результат – тот тип, с которым она определена; в противном случае сохраняется value category:
    - lvalue (в т.ч. переменная в скобках), результат – lvalue ссылка
    - xvalue (явная rvalue ссылка), результат – явная rvalue ссылка
    - prvalue, результат prvalue

decltype, примеры

- Начиная с C++14: decltype(auto) – это auto, но с правилами decltype

```
1.  int& foo();
2.  decltype(auto) i = foo(); // i is int&
3.  
4.  //...
5.  template<class F, class A>
6.  auto apply(F func, A&& a)
7.  // would give rvalue without ->decltype
8.  -> decltype(func(forward<A>(a)))
9.  {
10.         return func(forward<A>(a));
11. }
12. 
13. 
14. // or could be simplier
15. decltype(auto) apply(auto&& func, auto&&... args) {
16.     return func(forward<decltype(args)>(args)...);
17. }
```

Some `constexpr` abilities

constexpr metaprogramming (C++14)

Template metaprogramming	
1.	`template <size_t n>`
2.	`struct log_2`
3.	`{ const size_t value = 1 + log_2<n / 2>::value };`
4.	
5.	`template <>`
6.	`struct log_2<1>`
7.	`{ const size_t value = 1; };`

constexpr metaprogramming	
1.	`constexpr size_t log2(size_t n) {`
2.	`    return n < 2 ? 1 : 1 + log2(n / 2);`
3.	`}`
4.	
5.	`// variables, conditions, loops, other constexpr calls`
6.	`constexpr size_t factorial(size_t n){`
7.	`    size_t f = 1u;`
8.	`    for (size_t i = 1; i <=n; ++i) f *= i; // loops`
9.	
10.	`    return f;`
11	`}`

if constexpr (C++17)

- Появилась возможность делать compile-time if-проверки.
- Если условие не выполняется, содержимое не компилируется (развитие SFINAE).

```
1.  template <typename T>
2.  auto get_value(T t)
3.  {
4.      if constexpr (std::is_pointer_v<T>)
5.          return *t; // deduces return type to int for T = int*
6.      else
7.          return t;  // deduces return type to int for T = int
8.  }
```

Концепты (C++20)

Концепты

```
1. template<class I>
2. concept Integral = is_integral_v<I>;
```

- Задают явное требование на тип (не меняя сам тип)
- Требования на тип становятся частью интерфейса функции/класса
- Можно перегружать по этим требованиям функции и специализировать классы
- Многократно упрощают ошибки компиляции – сразу показывают несоответствие типов концепту
- Могут заменить auto (constrained placeholder)
- Есть много стандартных концептов, и можно делать свои

Определение и использование

Concept definition

```
1. template<class T>
2. concept Comparable = requires(T x) { x < x; };
3.
4. template<class T>
5. concept ComparableIt = requires(T x) { (*x) < (*x); };
```

Concept usage

```
1.  // usage
2.  template<class FwdIt>
3.      requires ComparableIt<FwdIt>
4.  void mysort(FwdIt begin, FwdIt end) { /*...*/ }
5.
6.
7.
8.
9.
10.
11.
```

Определение и использование

Concept definition

```
1. template<class T>
2. concept Comparable = requires(T x) { x < x; };
3.
4. template<class T>
5. concept ComparableIt = requires(T x) { (*x) < (*x); };
```

Concept usage

```
1.  // usage
2.  template<class FwdIt>
3.      requires ComparableIt<FwdIt>
4.  void mysort(FwdIt begin, FwdIt end) { /*...*/ }
5.
6.  // or like this
7.  template<ComparableIt FwdIt>
8.  void mysort(FwdIt begin, FwdIt end) { /*...*/ }
9.
10.
11.
```

Определение и использование

Concept definition

```
1. template<class T>
2. concept Comparable = requires(T x) { x < x; };
3.
4. template<class T>
5. concept ComparableIt = requires(T x) { (*x) < (*x); };
```

Concept usage

```
1.  // usage
2.  template<class FwdIt>
3.      requires ComparableIt<FwdIt>
4.  void mysort(FwdIt begin, FwdIt end) { /*...*/ }
5.
6.  // or like this
7.  template<ComparableIt FwdIt>
8.  void mysort(FwdIt begin, FwdIt end) { /*...*/ }
9.
10. // or even like this (instead of auto)
11. void mysort(ComparableIt begin, ComparableIt end){ /*...*/ }
```

Более сложные требования

Concept definition

```
1. template<class T, class Comp>
2. concept ComparableItBy = requires(T x, Comp c) {
3.     { c(*x, *x) } -> std::convertible_to<bool>;
4. };
```

Concept usage

```
1. template<class FwdIt, class Comp>
2.     requires ComparableItBy<FwdIt, Comp>
3. void mysort(FwdIt begin, FwdIt end, Comp c) { /*...*/ }
```

Сделаем перегрузку по свойству типа.

Сделаем перегрузку по свойству типа.

Как это было раньше?

Проверка на контейнер

- Можно проверить даже на наличие функции, полученной от базового класса. Для этого воспользуемся decltype.

```
1.  template <class T>
2.  struct has_begin_end
3.  {
4.      typedef char yes;
5.      typedef struct { yes dummy[2]; }no;
6.  
7.      template <typename U>
8.      static auto test(U* u)
9.          -> decltype((*u).begin(), (*u).end(), yes());
10.     static no   test(...);
11. 
12.     enum { value = (sizeof(yes) == sizeof test((T*)0)) };
13. };
```

Выбор по свойству типа enable_if

```
1.
2.  template<class T>
3.  enable_if_t<has_begin_end<T>::value> print(T const& cont)
4.  {
5.      for(auto const& item: cont)
6.          cout << item << " ";
7.  }
8.
9.  template<class T>
10. void print(
11.     T const& value, enable_if_t<!has_begin_end<T>::value>* = 0)
12. { cout << value; }
13.
```

- Тот самый SFINAE
- Не очень просто писать
- Не очень быстро компилировать (приходится сложно проверять и отбрасывать перегрузки)

Сделаем перегрузку по свойству типа.

А как это теперь!

Перегрузка по свойству типа

Concept definition

```
1. template<class C>
2. concept Iteratable = requires(remove_const_t<C> c) {
3.     { begin(c) }-> convertible_to<typename C::iterator>;
4.     { end  (c) }-> convertible_to<typename C::iterator>;
5. };
```

Concept usage

```
1. void print(Iteratable const& collection) {
2.     for (auto const& item : collection)
3.         cout << item << " ";
4. }
5.
6. void print(auto value) {
7.     cout << value << " ";
8. }
```

Модули (C++20)

Решаемая задача

```
1.  // main.cpp
2.  #include <iostream>
3.
4.  int main(int argc, char* argv[])
5.  {
6.      using namespace std;
7.      cout << "Hello, " << argv[1] << endl;
8.
9.      return 0;
10. }
```

- Хедера очень раздувают TU – долго компилировать
- В TU попадает все, что было в хедерах
  - Лишние макросы (проблема min/max)
  - Лишние объявления (поэтому в стандартной библиотеке такие странные имена в реализации)
- Можно случайно нарушить ODR, если макросом испортить включаемый после хедер
- В TU попадают все включенные в header другие header'а

Модули

```
1. import std.io
2. 
3. int main(int argc, char* argv[])
4. {
5.     std::cout << "Hello, " << argv[1] << std::endl;
6.     return 0;
7. }
```

- Позволяют
  - Ускорить сборку
  - Выборочно обозначить, что попадает в TU
  - Решить проблему случайного нарушения ODR
  - Отсечь попадание лишних зависимостей
- Особенности
  - Точка в std.io ничего не означает
  - Ортогональны пространствам имен
  - Строятся все символы из модуля один раз, имплементация компилируется отдельно
- Что не делают:
  - Никак не изменяют механизм распространения библиотек
  - Не улучшают, но и не ухудшают работу package manager'ов

my_sat_solver в header’e

```
1.  // my_sat_solver.h
2.  #pragma once
3.  #include "sat_solver.h“ // unwanted dependency header
4.
5.  namespace details
6.  {
7.      struct solver // unwanted declaration
8.      { /*...*/ sat_solver sol; }
9.
10.     template<class T>
11.     void do_solve(){
12.         /*...*/
13.     }
14. }
15.
16. template<class T>
17. bool solve(/*...*/) {
18.     details::do_solve<T>(/*...*/)
19. }
```

my_sat_solver в модуле

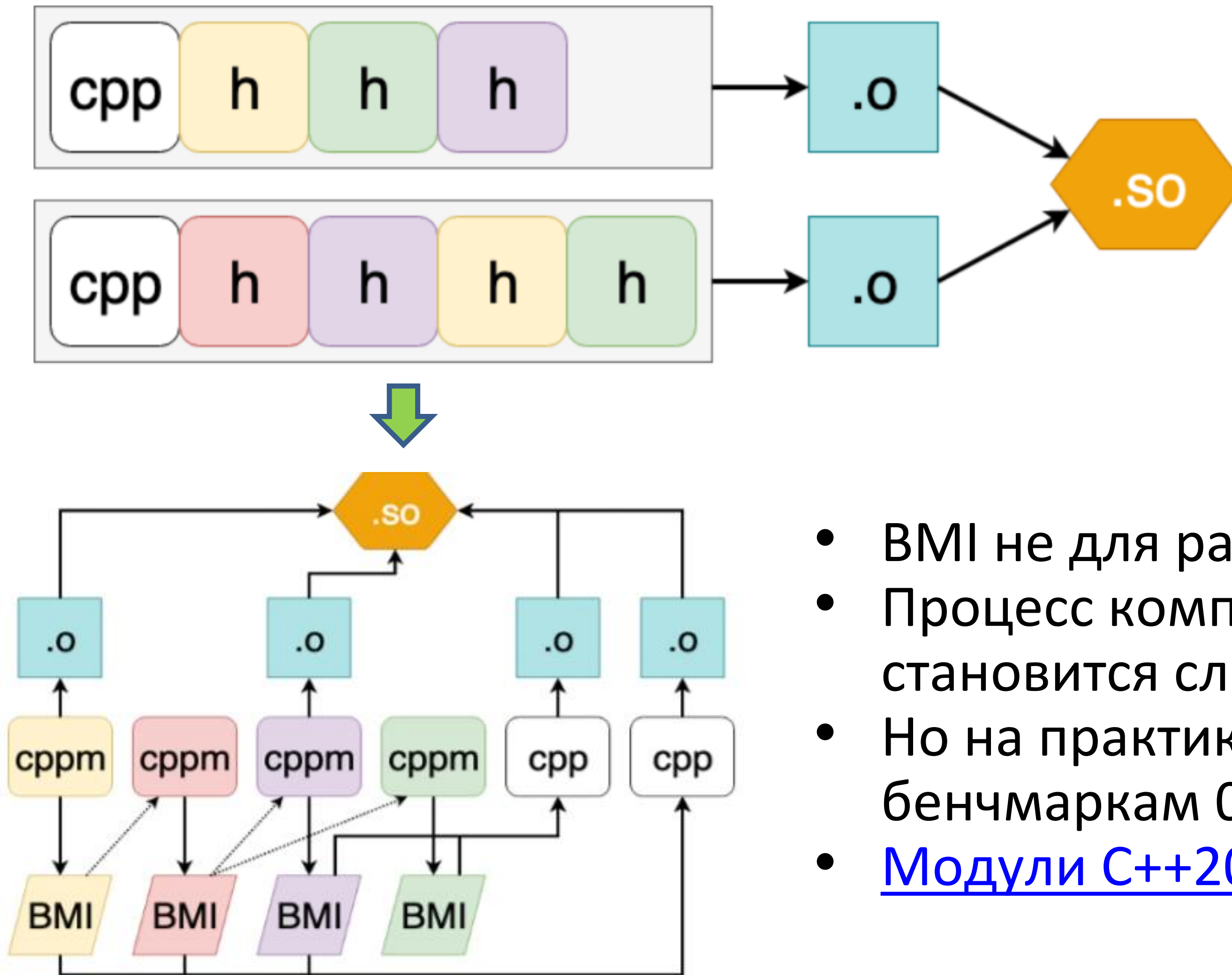
Module interface unit (my_sat_solver.cppm)

```
1.  export module my_sat_solver;
2.  struct solver;
3.
4.  template<class T>
5.  void do_solve(){
6.      /*...*/
7.  }
8.
9.  template<class T>
10. export bool solve(/*...*/) {
11.     do_solve<sat_solver>(/*...*/)
12. }
```

Module implementation unit (my_sat_solver_impl.cppm)

```
1.  module my_sat_solver;
2.  [export] import sat_solver; // it's possible to reexport
3.
4.  struct solver
5.  { /*...*/ sat_solver sol; }
```

Модель компиляции



- BMI не для распространения
- Процесс компиляция становится сложнее
- Но на практике быстрее. По бенчмаркам 0.5-10 раз
- Модули C++20 C++Russia.2019

Тернистый путь модулей

- Переходный период
  - Мало шансов переписать весь код на модули
- Header units
  - Рассматривать «чистые» хедера как модули
  - Все символы на export
  - Все макросы тоже попадут (к сожалению)
  - C++20: все хедера стандартной библиотеки – header unit

```
1. import some.module
2. import <vector>
```

Ranges (C++20), но есть в Boost

Boost.range. Range concept

- Функции std, оперируя итераторами, предоставляют большую гибкость, однако весьма “монструозны” в использовании.

```
1.  using namespace boost::adaptors;`
2.  std::vector<int> vi = ...;
3.  copy(vi | filtered(fn) | reversed, ostream_iterator<int>(cout));
```

- Концепция Range «легче» контейнеров:
  - не обязательно владеет элементами, к которым имеет доступ;
  - не обязательно обладает семантикой копирования.

Boost.range Алгоритмы и адапторы

- Алгоритмы дублируют (и даже расширяют) алгоритмы из STL.
  - возвращают снова Range для последовательного вызова.
- Адаптеры:
  - ленивы, нет лишних копий и аллокаций;
  - гибки, предоставляют последовательное обращение;
  - нет больше комбинаторного взрыва от _copy, _if.

Примеры

```
1.
2.  // 1
3.  boost::push_back(vec,
4.                  rng | replaced_if(pred, new_value) | reversed);
5.
6.  // 2
7.  std::map<int,int> input;
8.  boost::copy(input | map_keys,
9.              ostream_iterator<int>(cout, ","));
10.
11. // 3
12. std::vector<int> input;
13. boost::copy(input | filtered(fn) | reversed,
15.             ostream_iterator<int>(cout, ","));
16.
```

Вместо заключения…

Не пугайтесь, если не все понимаете. Это нормально!

Спасибо за внимание. Вопросы?