Лекция 7
Нововведения C++11/14.

Anonymous functions (C++11)

```
1.  // lambda functions
2.  [capture](params) [-> return-type] {body}
3.
4.  // samples
5.  [] (int x) { return x + global_y; }
6.  [] (int x) -> int
7.  { z = x + global_y; return z; }
8.
9.  // capture type
10. [x, &y](){} // capture x by value, y by ref
11. [=, &x](){} // capture everything by value, x by ref
12. [z = x + y](){ cout << z; }
13.
14. // more samples
15. matrix m;
16.
17. auto rot = [&m](point& p){ p *= m; }
18. for_each(points.begin(), points.end(), rot);
19. // or, shorter with help of boost.range library
20. boost::for_each(points, rot);
```

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

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

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. // or could be simplier
13. template<class F, class A>
14. decltype(auto) apply(F func, A&& a) {/*...*/}
15.
```

Forwarding problem

- А что же делать в случае многих параметров?

Variadic templates

- А что же делать в случае многих параметров?

```
1.  template<class T, class... Args>
2.  std::shared_ptr<T> factory(Args&&... args)
3.  {
4.      return std::shared_ptr<T>(
5.          new T(std::forward<Args>(args)...));
6.  }
7.
8.
9.  int main()
10. {
11.     auto x = factory<std::string>("str");
12.     return 0;
13. }
```

- Используется, когда нужно обработать схожим образом разнотипные параметры: std::bind, большинство фабрик, std::tuple, etc.

Parameter pack

- Variadic templates определяются через parameter pack:

```
1.  //-- in template declaration
2.
3.  // list of types
4.  class... Types
5.
6.  // list of templates
7.  template<param-list> class... Args
8.
9.
10. // list of values
11. Type... values
12.
13. //-- in function prototype, list of arguments
14. Types... args
15.
16. //-- pack expansion
17. pattern...
```

Примеры variadic templates

```
1.  template<class... Types>
2.  struct my_tuple {};
3.
4.  template<class Head, class... Types>
5.  struct my_tuple : my_tuple<Types> { /*...*/ Head value; };
6.
7.  template<class ... Types>
8.  void func(Types&&... args){/*...*/}
9.
10. auto func(auto&&... args){/*...*/}
11.
12. template<class... Bases>
13. struct T : public Bases...
14. {
15.     T(Bases const&... b) : Bases(b)...{}
16. };
17.
18. const size_t sz = sizeof...(args) + 1;
19. size_t res[sz] = {42, args...};
20.
21. auto func = [&, args...]{ bar(args...); };
```

Pack expansion & fold expression

```
1.  // expands to f(&a1, &a2, &a3)
2.  f(&args...);
3.
4.  // expands to f(n, ++a1, ++a2, ++a3);
5.  f(n, ++args...);
6.
7.  // f(const_cast<A1*>(&a1), const_cast<A2*>(&a2),
8.  const_cast<A3*>(&a3))
9.  f(const_cast<Args*>(&args)...);
10.
11. // f(h(a1,a2,a3) + a1, h(a1,a2,a3) + a2, h(a1,a2,a3) + a3)
12. f(h(args...) + args...);
13.
14. // fold expression
15. (args op ...) results in arg1 op(... op(argN - 1 op argN))
16. (... op args) results in ((arg1 op arg2) op ... ) op argN
17.
18. (args op ... op init) results in arg1 op(... op(argN op init))
19. (init op ... op args) results in ((arg1 op init) op ...) op argN
20.
```

Самостоятельно!

- Распечатайте содержимое tuple
  - Сперва сами сделайте рекурсию
  - Затем посмотрите на std::integer_sequence

Alias templates (gcc4.7)

- Появилась возможность делать typedef на шаблонный тип, очень удобно для enable_if

```
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 (vs11, gcc4.6)

- Намного компактнее итерирование по контейнеру, если не нужен индекс

```
1.  list<string> strings;
2.  for (list<string>::iterator it = strings.begin(); it !=
3.  strings.end(); ++it)
4.  {/*.*/}
5.  for (auto it = strings.begin(); it != strings.end(); ++it)
6.  {/*.*/}
7.  for (string& str: strings)
8.  {/*.*/}
9.  for (auto& str: strings)
10. {/*.*/}
```

nullptr constant (vs10, gcc4.6)

- nullptr может приводится к указателям, bool, но не к целым или floating-point

```
1.  void process(int);
2.  void process(const char*);
3.
4.  //..
5.  // nullptr could be cast to pointer or bool
6.  // not to int or double
7.
8.  process(0);      // what function will be called?
9.  process(nullptr);
```

initializer list (vs11, gcc4.4)

- 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<int> vi {4};
```

type traits (vs8, gcc4.3)

- Доступны через хидер <type_traits>.
- Определяются разнообразные свойства типов.
- Еще больше есть в boost type traits.

```
1.   is_const                                  is_integral
2.   is_standard_layout                        is_floating_point
3.   is_pod                                    is_class
4.   is_literal_type                           is_function
5.   is_polymorphic                           is_member_function_pointer
6.   is_abstract
7.                                            is_arithmetic
8.   is_constructible                         is_reference
9.   is_trivially_constructible
10.  is_nothrow_constructible
11.
12.  is_same
13.  is_base_of
14.  result_of
```

new string literals (vs11*, gcc4.5)

- 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(vs11, gcc4.7)

- 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. };
```

Inheriting Constructors (gcc4.8)

- Можно наследовать либо все конструкторы, либо ни одного.
- Не могут быть конструкторы в множественных базовых классах с одинаковым прототипом.
- Аналогично не может быть собственного конструктора с совпадающим по прототипу с базой конструктором.

```
1.  struct base
2.  {
3.      base(const char* str) {/*...*/}
4.  };
5.
6.  struct derived : base
7.  {
8.      derived(size_t number) {/*...*/}
9.      using base::base;
10. };
```

Non-static data member initializers(gcc4.7)

- 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. };
```

if constexpr (*)

- 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.  }
```

Спасибо за внимание. Вопросы?