Лекция 4. bind & function

Задача: вызов callback'а

```
1.  void set_timer(void(*)(double /*time*/), double /*time*/);
2.  void set_timer(void(*)(double, void* /*data*/), double /*time*/, void*);
3.
4.  template<class T> void set_timer(T* obj, void (T::*)(double), double);
5.  template<class F> void set_timer(F const& f, double/*time*/); // f(time)
6.
7.  void set_timer(ITimeHandler* handler, double /*time*/);
```

1. Не передать данные.
2. Передать, но не typesafe. Нужно формировать структуру.
3. Функций `set_timer` по количеству клиентов. Нужно иметь правильную функцию в типе `T`.
4. Можно и без правильной функции, но придется делать подходящий функтор.
5. Одна реализация, но требует полиморфный интерфейс от класса.

Немного магии. std::bind

- bind обобщает каррирование для функций нескольких аргументов.

```
1.  void foo(int a, string b, double c);
2.
3.  bind(foo, 1, "2", 3.)();                    // foo(1, "2", 3.)
4.  bind(foo, 1, _1 , 3 )("2");                 // foo(1, "2", 3.)
5.  bind(foo, _2, _3, _2)(2, 1, "2", 7);        // foo(1, "2", 1.)
6.
7.  auto bar = bind(foo, _2, _3, _2);
8.  bar(2, 1, "2", 7);
```

Как избежать копирования? ref/cref

- bind копирует переданные параметры. Но этого можно избежать.

```
1.  void foo(vector<string> const& vstr, int value) {/*...*/}
2.  //..
3.
4.  vector<string> v(1000);
5.  // makes copy
6.  auto a = bind(foo, v, _1);
7.  // no copy, used reference_wrapper
8.  auto b = bind(foo, cref(v), _1);
```

Использование bind с функторами

```
1.  struct func_t
2.  {
3.      string operator()(int    to_str) const{/*...*/}
4.      int    operator()(string to_int) const{/*...*/}
5.  };
6.  //..
7.
8.  func_t f;
9.
10. bind<int>   (f)("42"); // funct_t::operator()(string)
11. bind<string>(f)(24);   // funct_t::operator()(int)
```

- Перегрузка по-прежнему выполняется по переданному параметру.
- Если функтор предоставляет result_type (а-ля std::unary_function), то возвращаемый тип можно не указывать.

Вызов методов

```
1.  struct T
2.  {
3.      void func(string str) {/*...*/}
4.  };
5.
6.  T obj;
7.  shared_ptr<T> ptr(new T);
8.
9.  string name = "Edgar";
10.
11. bind(&T::func, ref(obj), _1)(name);
12. bind(&T::func, &obj    , _1)(name);
13. bind(&T::func, obj     , _1)(name);
14. bind(&T::func, ptr     , _1)(name);
```

- Вызовы 13-14 держат объект, пока существует сам биндер.
- Бинд виртуальных функций работает корректно.

Вложенность bind

- Как поступить, если параметр еще не зафиксирован, но его генератор известен?

```
1.  void foo(int x, int y);
2.  int  bar(int x);
3.
4.  auto func = bind(foo, bind(bar, _1), _2);
5.  func(a, b); // foo(bar(a), b)
```

- А можно ли не фиксировать саму функцию?

```
1.  int bar2(int x, int answer);
2.
3.  auto bb = bind(apply<int>(), _1, 42);
4.  bb(bar2, 5);
```

std::function

- Контейнер функции с фиксированным прототипом:
  - есть функции empty и clear;
  - можно использовать в условных выражениях.

```
1.  void foo(double a, double b) {/*...*/}
2.
3.  //..
4.  function<void(int, double)> f = foo;
5.  f(3, 14);
```

Синергический эффект bind & function

```
1.  typedef function<void(double/*time*/)> timer_f;
2.
3.  // just one function for all clients
4.  void set_timer(timer_f const& f, double time);
5.
6.  //..
7.  set_timer(bind(&window_t::redraw    , &wnd_)        , 10);
8.  set_timer(bind(&beeper_t::make_sound, &beeper_, 440), 10);
9.  set_timer(bind(&clock_t ::update    , &clock_ , _1) , 10);
```

- Всего одна нешаблонная (!) реализация.
- Допускаются дополнительные данные.
- Не требуется определять дополнительные функторы.
- Type safe.
- Нет необходимости в полиморфном интерфейсе и виртуальных функциях (простор для оптимизатора).

Быстродействие

- Дополнительное время уходит на выделения памяти. А, если их нет – на косвенные вызовы.
- `bind` не делает дополнительных выделений памяти. Косвенный вызов функции может быть оптимизирован в `bind` тогда же, когда и без него.
- `function` может выделять память под «большой» объект binder. Поэтому лишний раз не копируйте function.

Как работают placeholder'ы?

- Возможный путь реализации:

```
1.  template<class arg_t, class arg1_t, class arg2_t, class arg3_t>
2.  arg_t const& take_arg(arg_t  const& arg, arg1_t const&,
3.                        arg2_t const&    , arg3_t const&)
4.  {
5.      return arg;
6.  }
7.
8.  template<size_t N, class arg1_t, class arg2_t, class arg3_t>
9.  typename N_th<N, arg1_t, arg2_t, arg3_t>::type const&
10.     take_arg(placeholder<N> const& pl, arg1_t const& arg1,
11.              arg2_t const&       arg2, arg3_t const& arg3)
12. {
13.     return pl(arg1, arg2, arg3);
14. }
15.
16. template<class arg1_t, class arg2_t, class arg3_t>
17. auto binder_t::operator ()(arg1_t const& arg1, arg2_t const& arg2,
18.                           arg3_t const& arg3) -> R
19. {
20.     // K - number of func_ parameters
21.     func_(take_arg(pos1_, arg1, arg2, arg3),
22.           ... ,
23.           take_arg(posK_, arg1, arg2, arg3));
24. }
```

Вопросы?