Лекция 4.
Работа с событиями в современном C++

CS Club, Novosibirsk, 2019

Задача: вызов callback'а

```
1.  void set_timer(void(*)(double /*time*/));
2.  void set_timer(void(*)(double /*time*/, void* /*data*/), void*
    /*data*/);

    template<class T> void
3.    set_timer(double /*time*/, T* obj, void (T::*)(double));
4.  template<class F> void set_timer(double, F const& f); // f(time)

5.  void set_timer(double /*time*/, ITimeHandler* handler);
```

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 некоторый аналог каррирования 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.)
```

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);
```

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 b = bind(foo, bind(bar, _1), _2);
5.  b(x, y); // foo(bar(x), y)
```

- А можно ли не фиксировать саму функцию?

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

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 x, double y)> f = foo;
5.  f(3, 14);
```

Синергический эффект bind & function

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

- 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.
- Если часть программы не является bottleneck, то использование там связки bind/function, вероятнее всего, не ухудшит ситуацию.

Как могут работать 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. }
```

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.
13. // more samples
14. matrix m;
15.
16. auto rot = [&m](point& p){ p *= m; }
17. for_each(points.begin(), points.end(), rot);
```

Anonymous functions and bind

- Прежний пример с помощью lambda функций.

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

Some samples, part I

- События для перегруженной функции

```
1.
2.  struct some
3.  {
4.      void set_time(time_t time);
5.      void set_time(short hh, short mm, short ss);
6.      //..
7.  };
8.
9.  some s;
10. std::function<void(time_t)> f;
11.
12. f = bind(static_cast<void (some::*)(time_t)>(&some::set_time),
13.          &s, std::placeholders::_1);
14. // or
15. f = [&s](time_t t){s.set_time(t);};
```

Some samples, part II

- События для функций многих аргументов

```
1.
2.  struct logger_t
3.  {
4.      virtual void log(string filemame, unsigned line, string what){}
5.      // ...
6.  };
7.
8.  struct file_logger : logger_t{/*...*/};
9.
10. //..
11. file_logger fl;
12.
13. add_log(bind(&logger_t::log, &fl, _1, _2, _3));
14. // or
15. add_log([&fl](string filename, unsigned line, string what)
16.         { fl.log(filename, line, what); });
17.
```

Bind vs Lambda

- когда Bind
  - + Много (или длинные) параметры функций
  - + Выражение из одного вызова

- когда Lambda
  - + Сложное выражение или их несколько
  - + Перегрузка функции
  - + Покрывает весь функционал bind (но не наоборот), но порой исходный код с bind более выразителен

Спасибо за внимание!
Вопросы?