Лекция 11. Введение в шаблоны. Часть 2.

Специализация. Связь с наследованием. Умные указатели

Шаблонный класс

```
1.  template<class T>
2.  struct vector
3.  {
4.      vector(size_t count = 0, T def = T());
5.      vector(vector const& other);
6.      // don't need to write vector<T> inside
7.      vector& operator=(vector const& other);
8.
9.      void resize(size_t size);
10.     void clear ();
11.
12.     T&       operator[](size_t index);
13.     T const& operator[](size_t index) const;
14.     size_t size() const;
15.
16. private:
17.     T*      data_;
18.     size_t  size_;
19. };
```

Специализация шаблонов класса

- Можно указать особую реализацию класса для выбранного типа
- Такая специализация не обязана иметь тот же интерфейс, что и общий шаблон (хотя это обычно ожидается пользователями)

```
1.  template<class T>
2.  struct vector
3.  {
4.  /* impl */
5.  };
6.
7.  template<>
8.  struct vector<bool>
9.  {
10. /* compact storage for bool array
11.    e.g. in bits */
12. };
```

Частичная специализация

- Допустима только для классов. Функции требуют полной специализации.

```
1.  template<class T, class A>
2.  struct vector
3.  {
4.  /* impl */
5.  };
6.
7.  template<class A>
8.  struct vector<bool, A>
9.  {
10. /* compact storage for bool array
11.    e.g. in bits */
12. };
```

Общая нешаблонная база

- Позволяет значительно сэкономить на кодогенерации.

```
1.  template<class T>
2.  struct vector<T*>
3.      : private vector<void*>
4.  {
5.      typedef vector<void*> base_t;
6.      typedef T*            value_type;
7.
8.      explicit vector(size_t size,
9.                      value_type def = value_type())
10.         : base_t(size, def)
11.     {}
12.
13.     T& at(size_t index)
14.     { return static_cast<value_type>(base_t::at(index)); }
15.
16.     T& operator[](size_t index) { return at(index); }
17. };
```

Явное инстанцирование

- Позволяет избежать многократного инстанцирования одного и того же шаблона

```
1.  //******** string.h ******************/
2.
3.  /* a little more complex in real STL */
4.  template<class char_t>
5.  class basic_string{/*...*/};
6.
7.  typedef basic_string<char> string;
8.
9.  // prevent of instantiation
10. extern template class basic_string<char>;
11.
12.
13. //******** some.cpp or lib ************/
14. // explicit instantiation
15. template class basic_string<char>;
```

Специализация шаблонов функций

- Допускается только полная специализация. Но и она-то редко нужна.

```
1.  template<class T>
2.  void swap(T& x, T& y)
3.  {
4.      T tmp = x;
5.      x = y;
6.      y = tmp;
7.  }
8.
9.  template<>
10. void swap<my_class>(my_class&, my_class&)
11. {
12.     /* effective implementation */
13. };
```

Специализация шаблонов функций

- Допускается только полная специализация. Но и она-то редко нужна – если передается параметр типа, хватит перегрузки.

```
1.  // overriding
2.  void swap(my_class&, my_class&)
3.  {
4.      /* effective implementation */
5.  };
6.
7.  template<class T>
8.  void swap(vector<T>& x, vector<T>& y)
9.  {
10.     using std::swap; //for all the functions named swap(!)
11.     swap(x.data_, y.data_);
12.     swap(x.size_, y.size_);
13. };
```

Curiously recurring template pattern (CRTP)

```
1.  template<class derived_t>
2.  struct comparable
3.  {
4.      bool operator==(derived_t const&) const;
5.      bool operator!=(derived_t const&) const;
6.
7.      bool operator>=(derived_t const& rhs) const
8.      { rhs < derived(); }
9.
10. private:
11.     derived_t&       derived()
12.     {return static_cast<derived_t&>(*this);}
13.     derived_t const& derived() const
14.     {return static_cast<derived_t const&>(*this);}
15. };
16.
17. struct point
18.     : comparable<point>
19. { bool operator<(point const& rhs) const; /**/};
20.
21. //--------------------
22. point x(...), y(...);
23. bool check = x != y;
```

Наследование и шаблоны

- Шаблоны от динамически полиморфных типов не связаны между собой

```
1.  struct base {/**/};
2.  struct derived : base {/**/};
3.
4.  //
5.  vector<base> vb;
6.  vector<derived> vd;
7.
8.  vb = vd; // error
9.
10. // but
11. vector<base*> vpb;
12. vpb.push_back(new derived);
```

Шаблонные функции в нешаблонном классе

```
1.  struct any
2.  {
3.      template<class T>
4.      any(T const& obj);
5.
6.      any(any const& other);
7.
8.      /*....*/
9.  };
```

- Такие функции не могут быть виртуальными
- Но! Шаблонный класс может реализовывать абстрактный интерфейс
- Осторожно: шаблонный конструктор не замещает генерируемый

Как связать шаблоны класса, если связаны его параметры?

```
1.  template<class T>
2.  struct shared_ptr
3.  {
4.      shared_ptr() : ptr_(nullptr), count_(0){}
5.      shared_ptr(shared_ptr const&){/**/}
6.
7.      template<class U>
8.      shared_ptr(shared_ptr<U> const& other)
9.          : ptr_  (other.get())
10.         , count_(other.count_)
11.     { ++(*count_); } /*a little more complex in real life*/
12.     /*....*/
13.     T* get() const { return ptr_; }
14. private:
15.     T*      ptr_;
16.     size_t* count_;
17. };
18. //---------------------------
19. shared_ptr<derived> pd;
20. shared_ptr<base> pb(pd);
```

Динамический vs Статический полиморфизм

- Один и тот же код используется для в действительности разных типов
- Может использоваться для типов, неизвестных на момент написания кода

- **Динамический**:
  - Единообразная работа, основанная на отношении надмножество/подмножество
  - Динамическое связывание и раздельная компиляция
  - Необходима бинарная согласованность (ABI)

Динамический vs Статический полиморфизм

- **Статический**:
  - Единообразная работа, основанная на синтактическом и семантическом интерфейсе
  - Статическое связывание. Мешает раздельной компиляции
  - Высокая эффективность (такая же, как и специальный код для типа)

Пара слов про exceptions

```
1.  struct my_exception
2.      : std::runtime_error
3.  {
4.      my_exception(const char* my_description)
5.          : std::runtime_error(my_description)
6.      {}
7.  };
8.
9.  try
10. {
11.     /*...*/
12.     if (bad_condition)
13.         throw my_exception("bad_condition");
14. }
15. catch (std::exception const& err) // (!) only ref
16. {
17.     std::cerr << err.what();
18. }
```

Умные указатели

- Почти те же указатели, только умнее
  - представляют собой RAII классы
  - поддерживают тот же интерфейс, что и обычные указатели: `op->`, `op*`, `op<` (например, чтобы положить в `std::set`)
  - сами управляют временем жизни объекта – вовремя вызывают деструкторы и освобождают память

Польза умных указателей

- Автоматическое освобождение памяти при удалении самого указателя
- Безопасность исключений

```
1.  void foo()
2.  {
3.      shared_ptr<my_class> ptr(new my_class("arg"));
4.      // or shorter definition:
5.      auto ptr = make_shared<my_class>("arg");
6.
7.      ptr->bar(); // if throws exception, nothing bad happened
8.  }
9.
10. void foo()
11. {
12.     my_class* ptr = new my_class(/*...*/);
13.     ptr->bar(); // oops, troubles in case of exception
14.     delete ptr; // common trouble is to forget delete
15. }
```

Популярные умные указатели

- **`std::scoped_ptr`**
- `std::unique_ptr`
- **`std::shared_ptr`**
- `std::weak_ptr`
- `boost::intrusive_ptr`

- Deprecated: `std::auto_ptr`

scoped_ptr

- Удобен для хранения указателя на стеке или полем класса. Не позволяет копироваться.

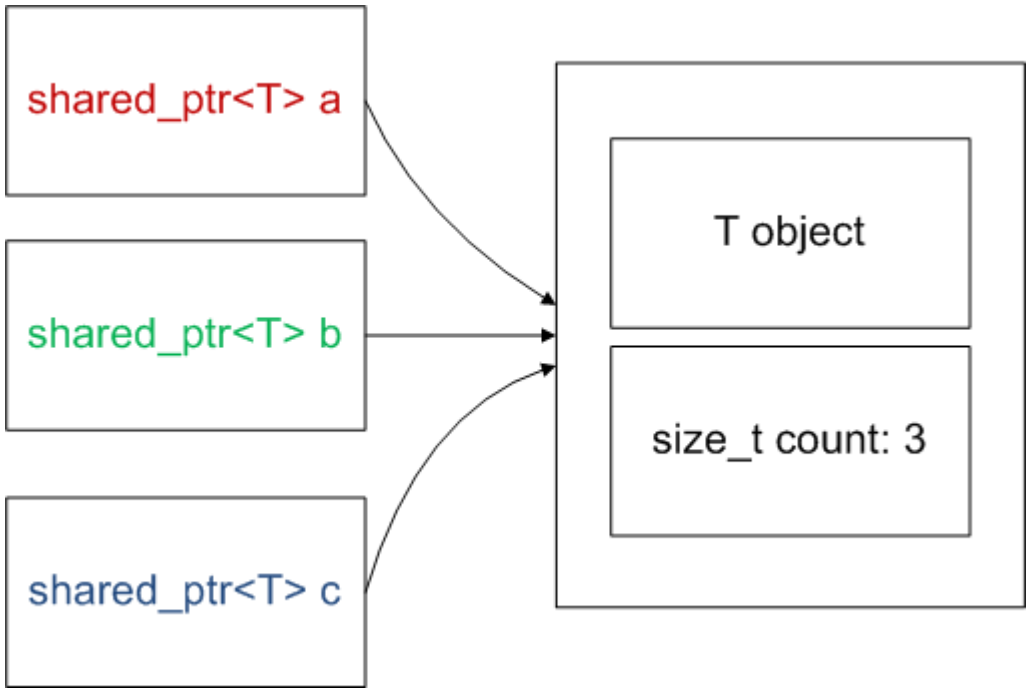
```
 1. template<class T> struct scoped_ptr : noncopyable {
 2.
 3. public:
 4.     typedef T element_type;
 5.
 6.     explicit scoped_ptr(T * p = 0);
 7.     ~scoped_ptr();
 8.
 9.     void reset(T * p = 0);
10.
11.     T & operator *() const;
12.     T * operator->() const;
13.     T * get() const;
14.
15.     operator unspecified-bool-type() const;
16. };
17. //-------
18. scoped_ptr<int> p(new int(5));
```

Почему explicit конструктор?

```
1.  //what if scoped_ptr had implicit constructor
2.  void foo(scoped_ptr<my_class> ptr)
3.  {
4.      /*...*/
5.  }
6.
7.  auto p = new my_class(/*...*/);
8.  foo(p);       // epic fail, p is not valid after this call
9.
10. p->do_smth();// error
11. delete p;    // one more error
```

shared_ptr

- Поддерживает общий счетчик ссылок на выделенный объект
- Удаляет объект только, когда последний из ссылающихся shared_ptr’ов удаляется или принимает указатель на другой объект



shared_ptr

- Наиболее используемый.
- Удобен для разделения владением.
- Можно возвращать из функций.
- *Можно передавать между модулями - запоминает правильную функцию удаления (из нужной библиотеки)

```
1.  template<class T> struct shared_ptr
2.  {
3.      /* more than scoped_ptr has */
4.      shared_ptr(shared_ptr const & r);
5.      template<class Y> shared_ptr(shared_ptr<Y> const & r);
6.
7.      shared_ptr(shared_ptr && r);
8.      template<class Y> shared_ptr(shared_ptr<Y> && r);
9.
10.     bool unique() const;
11.     long use_count() const;
12.     /*...*/
13. };
```

Вопросы?