Шаблоны

ЛЕКЦИЯ №7

Является ли тип указателем?

Специализация шаблонов

```
template <class T> struct is_pointer{
enum {Value = false};
};

template <class T> struct is_pointer<T*>{
enum {Value = true};
};
```

```
std::cout << is_pointer<int*>::Value?"Pointer":"Not pointer";
```

enum

наследие C

```
enum Days { Saturday,Sunday,Tuesday,Wednesday,Thursday,Friday};
Days day = Saturday;

if(day == Saturday)
  { std::cout<<"Ok its Saturday"; }
```

Variadic template

Example57_VariadicTemplate

```
template <class T> void print(const T& t) {
std::cout << t << std::endl;
}

template <class First, class... Rest>
void print(const First& first, const Rest&... rest) {
std::cout << first << ", ";
print(rest...); // рекурсия на стадии компиляции!
}
```

Variadic template в структурах данных
Example57_VariadicTemplate2

```
1.// Конец рекурсии
2.template <class... Ts> class tuple {};
3.// Шаблон
4.template <class T, class... Ts>
5.// Класс наследник самого себя но с меньшим числом параметров
6.class tuple<T, Ts...> : public tuple<Ts...> {
7.  public:
8.   tuple(T t, Ts... ts) : tuple<Ts...>(ts...), value(t) {}
9.// Ссылка на родителя
10.   tuple<Ts...> &next = static_cast<tuple<Ts...>&>(*this);
11.// Последний параметр
12.   T value;
13.};
```

Что внутри?

```
1.class tuple<double, uint64_t, const char*> : public tuple<uint64_t, const char*>
  {
2.  double value;
3.}

4.class tuple<uint64_t, const char*> : public tuple<const char*> {
5.  uint64_t value;
6.}

7.class tuple<const char*> : public tuple {
8.  const char* value;
9.}

10.class tuple {
11.}
```

std::enable_if

```
1.      // Шаблон
2.      template<bool, typename _Tp = void>
3.      struct enable_if { };

4.      // Специализация шаблона, в случае если параметр - истина
5.      template<typename _Tp>
6.      struct enable_if<true, _Tp>{
7.     typedef _Tp type;
8.};
```

Вспомогательный тип

```
1.// специальная структура для определения типа конкретного элемента в
  tuple
2.template <size_t, class> struct elem_type_holder;

3.// без параметра - это тип базового класса
4.template <class T, class... Ts> struct elem_type_holder<0, tuple<T,
  Ts...>> {
5.  typedef T type; // тип
6.};

7.// это тип k-го класса в цепочке наследования
8.template <size_t k, class T, class... Ts> struct elem_type_holder<k,
  tuple<T, Ts...>> {
9.typedef typename elem_type_holder<k - 1, tuple<Ts...>>::type type;
10.};
```

Шаблонная функция get
Example57_VariadicTemplate2Full

```
1.// шаблон функции get для получения параметра (данная специализация работает только при
  index==0)
2.template <size_t index,class ...Ts>
3.typename std::enable_if<index == 0,
4. typename elem_type_holder<0, tuple<Ts...>>::type&>::type
5.get(tuple<Ts...>& t){
6.    return t.value;
7.}
8.// шаблон функции get для получения параметра (данная специализация работает только при
  index!=0)
9.template <size_t index,class T,class ...Ts>
10.typename std::enable_if<index != 0,
11.   typename elem_type_holder<index, tuple<T,Ts...>>::type&>::type
12.get(tuple<T,Ts...>& t){
13.    tuple<Ts...> &base = t.next;
14.    return get<index-1>(base);
15.}
```

CRTP (Curiously Recurring Template Pattern)

Example60_CRTP

```
 template <class T>
class base{};

class derived : public
base<derived> {};
```

Такая конструкция делает возможным обращение к производному классу из базового!

Множественное наследование в шаблонах
Example58_VariadicTemplate3

```
1.template <typename... BaseClasses>
2.class Printer : public BaseClasses... {
3.public:
4.Printer(BaseClasses&&... base_classes) :
 BaseClasses(base_classes)...
5.{
6.}
7.};
```

Шаблоны в качестве параметров шаблонов

Example59_TemplateParameter

- Шаблон можно указать в качестве параметра шаблона!
- Все типы, которые используются при конструировании нового типа с помощью шаблона – должны быть его параметрами.

```
template <class T> class Payload
{
  …
};

template <template <class> class PL, class T> class Printer
{
…
};
Printer<Payload, int> printer;
```

Что нового в C++: auto

`Example62_Auto`

В C++11 auto позволяет не указывать тип переменной явно, говоря компилятору, чтобы он сам определил фактический тип переменной, на основе типа инициализируемого значения. Это может использоваться **при объявлении переменных** в различных областях видимости, как, например, пространство имен, блоки, инициализация в цикле и т.п.

```
auto i = 42;       // i - int
auto l = 42LL;     // l - long long
auto p = new foo(); // p - foo*
```

Использование auto позволяет сократить код (если, конечно, тип не int, который на одну букву меньше).

Не может использоваться в объявлении параметров функции или класса;

Протечка абстракции

Неудобной составляющей работы с коллекциями объектов родительского типа является необходимость приведения родительского типа к типу-наследнику (для выполнения необходимых операций над элементом коллекции). Т.е. мы жертвуем статическим контролем типов.

Протеска абстракции номер 1 Example66_AbstractionLeak1

```
class Figure{
public:
virtual double Square()=0;};


class Circle : public Figure{
public:
double Square() override{
        return 3.14*3.14*R;};};


class Sphere : public Circle{
public:
double Volume() {
        return 3.14*3.14*3.14*R;    };
   };
```

```
 // abstraction leak
for(Figure *figure:array)
    {
       Sphere *sphere = dynamic_cast<Sphere*> (figure);
       if(sphere!=nullptr)
       std::cout << "Volume:"
               << sphere->Volume()
        << std::endl;
  }
```

Протечка абстракции 2
Example67_AbstractionLeak2

```
1.class Figure{
2.public:
3.    virtual double Square()=0;
4.    virtual double Volume() { return 0.0;};
5.    virtual ~Figure() {};
6.};
```

```
7. Figure* array[]={new Circle(1),new Circle (2),new Sphere(1)};
8. // ISP (Interface Segregation Principle) fail
9.  for(Figure *figure:array) std::cout << "Square:" << figure->Square() << std::endl;
10. for(Figure *figure:array)  std::cout << "Volume:" << figure->Volume() << std::endl;
```

Используем tuple
Example68_AbstractionLeak3

```
1. tuple<Circle,Circle,Sphere,Sphere>
2.   t(Circle(1),Circle(2),Sphere(1),Sphere(2));


3.// в параметры get<size_t> можно подставить только константу
4. std::cout << "Square:" << get<0>(t).Square() << std::endl;
5. std::cout << "Volume:" << get<3>(t).Volume() << std::endl;
```

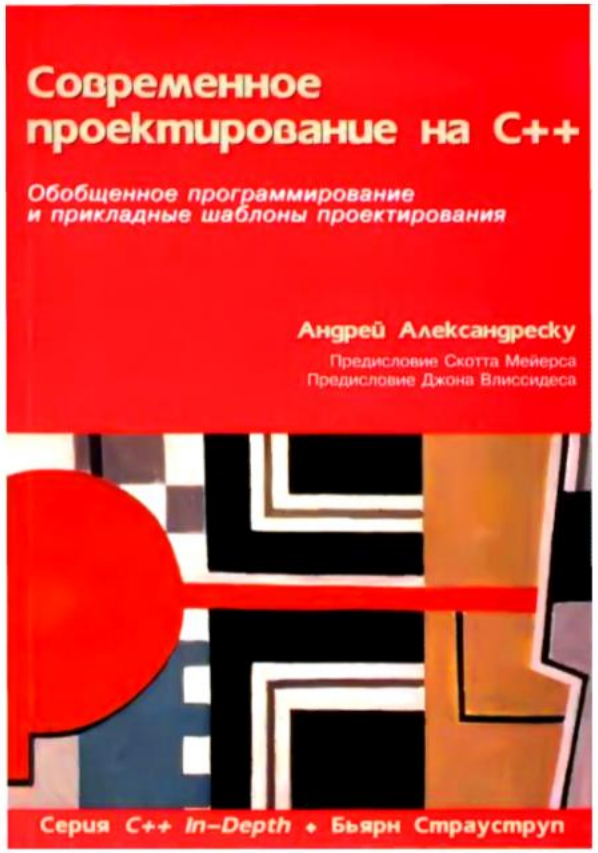
template alias
Example69_TemplateAlias

```
1.template <class A,class B> class Pair {
2.    public:
3.        A a;
4.        B b;
5.        Pair(A v1,B v2) : a(v1), b(v2) {};
6.};

7.template <class A> using SamePair =  Pair<A,A>;

8.int main(int argc, char** argv) {
9.    Pair<int,const char*> a(1,"one");
10.   SamePair<int> b(2,2);
11.
12.   return 0;
13.}
```

Книга про шаблоны



Год выпуска: 2002
Автор: Андрей Александреску
Жанр: Программирование [C++]
Издательство: Издательский дом "Вильямс" Москва - Санкт-Петербург - Киев
ISBN: 5-8459-0351-3(рус), 0-201-77581-6(англ.)
Количество страниц: 326
Описание: В книге изложена новая технология программирования, представляющая собой сплав обобщённого программирования шаблонов и объектно - ориентированного программирования на C++. Обобщённые компоненты, созданные автором, высоко подняли уровень абстракции, наделив язык C++ чертами языка спецификации проектирования, сохранив всю его мощь и выразительность
В книге изложены способы реализации основных шаблонов проектирования. Разные компоненты воплощены в библиотеке Loki, которую можно загрузить с Web-страницы автора. Книга предназначена для опытных программистов на C++

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ