Шаблоны

ЛЕКЦИЯ №7

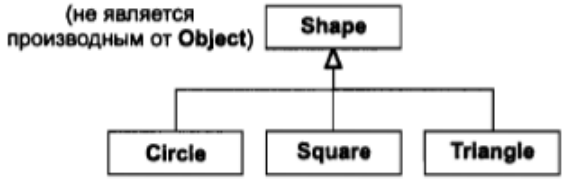
Два вида многократного использования кода

Наследование

- Создаем структуру для работы с «базовым классом»
- Создаем классы-наследники на каждый случай.

Шаблоны

- Описываем «стратегию работы» с «неопределенным» классом.
- Компилятор в момент создание класса по шаблону, сам создает нужный «код» для конкретного класса.



Template это …

- Шаблон – это параметрическая функция или класс.
- Параметром может являться как значение переменной (как в обычных функциях) так и тип данных.
- Параметры подставляются на этапе компиляции программы.
- Подставляя параметры в шаблон – мы конструируем новый тип данных (или функцию, если это шаблон функции)

Простой шаблон
Example50_Template

```
template <class T> class Print
{
 public:
 Print(T value)
 {
 std::cout << "Value:" << value << std::endl;
 };
};
```

Перед описанием класса ставим ключевое слово template <class T>

T – используем вместо имени класса, который будет заменяться при создании конкретного экземпляра класса.

`Print` – это шаблон

`Print<int>` - это класс, сконструированный по шаблону

Несколько параметров и шаблоны-функции

Example51_MultiTemplate

Параметры указываются через запятую:

```
template <class A, class B> class Sum { …}
```

Оператор, принимающий в качестве параметра – шаблон с параметрами:

```
template <class A, class B> std::ostream& operator<<(std::ostream & os, Sum<A, B> &sum)
```

Параметры – переменные
Example52_ComplexParameters

```
1.template <class TYPE, TYPE def_value, size_t SIZE = 10 > class Array {
2.protected:
3.    TYPE _array[SIZE];
4.public:
5.    Array() {
6.        for (int i = 0; i < SIZE; i++) {
7.            _array[i] = def_value;
8.        }
9.    }
10.    const size_t size() {
11.        return SIZE;
12.    }
13.    const TYPE operator[](size_t index) {
14.        if ((index >= 0) && (index < SIZE)) return _array[index];
15.        else throw BadIndexException(index, SIZE);
16.    }
17.};
```

Специализация шаблонов

Example53_TemplateSpecialization

```
template <class T>
class mycontainer {
// …
};
template <>
class mycontainer <char> {
// ..
};
```

Иногда бывает необходимость сделать специальную реализацию шаблона для какого-либо типа.

В этом случае, можно описать отдельную реализацию класса, дополнив его новыми методами или переопределив реализацию существующих.

Можно специализировать только часть параметров

Example54_TemplateSpecialization2

```
template <class A, class B,
class C> class Sum {

…

}
```

```
template <class A, class B>
class Sum<A, B, const char*> {

…

}
```

При частичной специализации у шаблона становится меньше параметров (какие-то мы уже указали явно).

Частичная специализация работает только с классами (с функциями не работает).

Вычисляем факториал
Example54_Factorial

```
1.// факториал с помощью функций
2.template <uint64_t value> uint64_t Factorial(){
3.  return Factorial<value-1>()*value;
4.}
5.template <> uint64_t Factorial<0>(){
6.  return 1;
7.}
8.// факториал с помощью классов
9.template<uint64_t n>class fact{
10.  public:
11.   static const uint64_t value = fact<n-1>::value * n;
12.};
13. template<>class fact<0>{
14.   public:
15.   static const uint64_t value = 1;
16.};
```

Templates
две модели

1. Наиболее популярный подход - модель включения(**inclusion model**), определения шаблонов полностью размещаются в заголовочном файле.
2. Модель явного инстанцирования (**explicit instantiation model**), как правило реализуется директивой явного инстанцирования (explicit instantiation directive).

Inclusion model

```
template<class T> class stack {
T* v;
T* p;
int sz;
public:
stack(int s) { v = p = new T[sz=s]; }
~stack() { delete[] v; }
void push(T a) { *p++ = a; }
T pop() { return *--p; }
int size() const { return p-v; }
};
```

И объявление и описание шаблона располагается в header файле (.h)

Фактически, при любом подключении к .cpp файлу – это будет новый шаблон для компилятора.

Минус такой модели в том, что трудно читать код (все перемешано).

explicit instantiation model
Example55_ExplicitInstantiation

```
template <class T> class
MyStack
{
public:
MyStack(void);
};
```

```
template <class T>
MyStack<T>::MyStack(void)
{
_size = 0;
_current = NULL;
}
template class MyStack<class
MyClass>;
```

В продолжение примера

1. В качестве параметра шаблона можно передавать указатели на функции (если работать не с указателями – то это уже будет вызов функции ☺)
2. В примере «параметр-функция» нам понадобился что бы удалять указатели. Если бы мы в коде написали «delete old->item» то такой код не скомпилировался бы для класса MyStack<MyClass>.
3. А вот для MyStack<MyClass*> - скомпилировался бы.

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ