Лекция 11. Шаблоны и обработка исключений

Казахский национальный университет имени аль-Фараби

Кафедра компьютерных наук

Объектно-ориентированное программирование

Лектор: старший преподаватель, PhD, Кенжебек Ержан Ғалымжанұлы

Цель лекции

Цель лекции

Изучить шаблоны функций и классов как средство обобщённого программирования и научиться писать код, не зависящий от конкретных типов данных.

Ожидаемые результаты

1. Писать шаблоны функций
2. Объяснять, как компилятор создаёт код по шаблону
3. Определять шаблоны классов (например, Stack)
4. Создавать объекты шаблонного класса для разных типов

План лекции

- Введение
- Шаблоны функций
- Синтаксис шаблонов
- Шаблоны классов

Введение

- В этой лекции рассматриваются два важных инструмента C++ — шаблоны и исключения. Шаблоны позволяют использовать одни и те же функции или классы для обработки данных разных типов. Исключения дают возможность обрабатывать ошибки, возникающие в классах, удобным и универсальным способом.
- Концепция шаблонов может быть использована в двух видах: по отношению к функциям и по отношению к классам. Сначала мы взглянем на шаблоны функций, затем на шаблоны классов, а закончим лекцию разговором о механизме исключений.

Шаблоны функций

- Допустим, вам нужно написать функцию вычисления модуля чисел.

```
int abs(int n)                  // вычисление модулей целых чисел
{
   return (n < 0) ? -n : n;   //если отрицательное, вернуть -n
}
```

- Определенная выше функция берет аргумент типа int и возвращает результат того же типа. Но теперь представьте, что понадобилось найти модуль числа типа long. Придется писать еще одну функцию

Шаблоны функций

```
long abs(long n)            // вычисление модулей чисел типа long
{
   return (n < 0) ? -n : n;
}
```

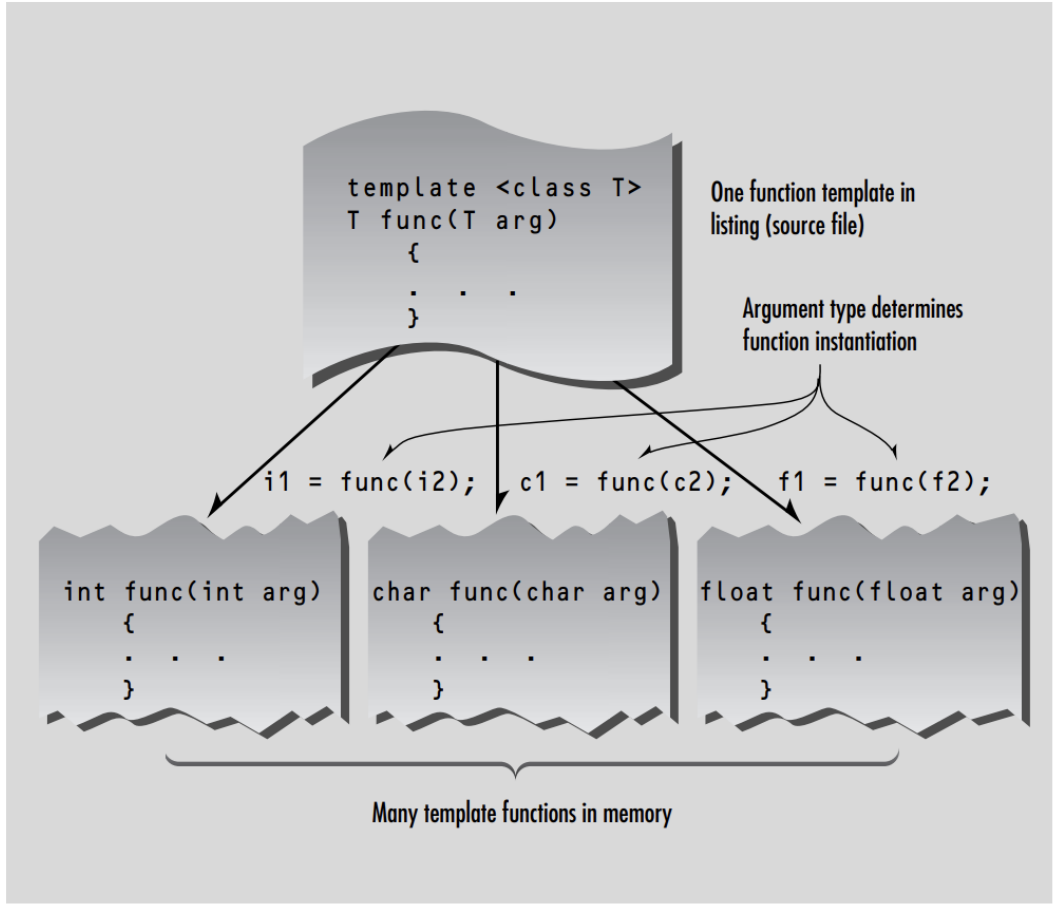
Да, теперь еще одну для float:

```
float abs(float n)          // модуль целых чисел
{
   return (n < 0) ? -n : n; // если отрицательное, вернуть -n
}
```

Тело функций, как видите, ничем не отличается. И все же они совершенно разные, поскольку обрабатывают аргументы и возвращают значения разных типов. Они могут быть перегружены и иметь одинаковые имена, это правда, но все равно для каждой из них нужно писать отдельное определение.

Мотивация использования шаблона

- Было бы здорово, если бы был какой-то способ написать эту функцию всего один раз и заставить ее работать с различными типами данных, возвращая, соответственно, результаты разного типа. Это как раз то, для чего существуют шаблоны функций в C++. Эта простая и гениальная идея схематично изображена



Шаблон функций (реализация), simpleTemplate.cpp

```cpp
// Шаблон функции вычисления модуля числа
#include <iostream>
using namespace std;
//-----------------------------------------------------------
template <class T>           // Шаблон функции!
T abs(T n)
{
    return (n < 0) ? -n : n;
}
//-----------------------------------------------------------
int main()
{
  int int1 = 5;
  int int2 = -6;
  long lon1 = 70000L;
  long lon2 = -80000L;
  double  dub1 = 9.95;
  double  dub2 = -10.15;

  // осуществления вызовов
  cout << "\nabs(" << int1 << ")=" << abs(int1); // abs(int)
  cout << "\nabs(" << int2 << ")=" << abs(int2); // abs(int)
  cout << "\nabs(" << lon1 << ")=" << abs(lon1); // abs(long)
  cout << "\nabs(" << lon2 << ")=" << abs(lon2); // abs(long)
  cout << "\nabs(" << dub1 << ")=" << abs(dub1); // abs(double)
  cout << "\nabs(" << dub2 << ")=" << abs(dub2); // abs(double)
  cout << endl;
  return 0;
}
```

Результаты работы программы:

```
abs(5)= 5
abs(-6)= 6
abs(70000)= 70000
abs(-80000)= 80000
abs(9.95)= 9.95
abs(-10.15)= 10.15
```

- Вся конструкция, начиная от ключевого слова template в начале первой строки и включая определение функции, называется шаблоном функции. Каким образом достигается такая потрясающая гибкость использования функции?

```
template <class T>      // шаблон функции
T abs(T n)
{
  return (n < 0) ? -n : n;
}
```

Синтаксис шаблона

- Краеугольным камнем концепции шаблонов функции является представление использующегося функцией типа не в виде какого-либо специфического (например, int), а с помощью названия, вместо которого может быть подставлен любой тип. В приведенном примере таким именем является T (никакого сакрального смысла именно в таком названии нет. На его месте может все что угодно, например anyType или YooHoo).
- Ключевое слово template сообщает компилятору о том, что «сейчас будет море крови»: мы определяем шаблон функции. Ключевое слово class, заключенное в угловые скобки, можно с тем же успехом заменить на type. Как вы уже видели, можно определять собственные типы, используя классы, поэтому разницы между типами и классами в известном смысле нет совсем. Переменная, следующая за словом class (T в нашем примере), называется аргументом шаблона. Внутри всего определения шаблона любой конкретный тип данных, такой, как int, заменяет аргумент T. В abs() это имя встречается лишь дважды в первой строке в качестве типа аргумента и одновременно в качестве типа функции. В более сложном случае оно могло бы встретиться много раз, в том числе в теле функции.

Как компилятор работает при шаблоне

- Генерации кода не происходит до тех пор, пока функция не будет реально вызвана в ходе выполнения программы. В примере simpleTemplate.cpp это происходит в конструкциях типа abs(int) в выражении:

```
cout << "\nabs(" << int1 << ")=" << abs(int1);
```

- Когда компилятор видит такой вызов функции, он знает, что нужно использовать тип int, поскольку это тип аргумента int1, переданного в шаблон. Поэтому он генерирует код abs для типа int, подставляя его везде вместо T. Это называется **реализацией** шаблона функции; при этом каждый реализованный шаблон функции называется **шаблонной функцией**

Шаблоны классов

- Шаблонный принцип можно расширить и на классы. В этом случае шаблоны обычно используются, когда класс является хранилищем данных (хороший тому пример мы увидим в следующей лекции «Стандартная библиотека шаблонов (STL)»).
- Стеки и связные списки, рассмотренные нами в предыдущих главах, — тоже типичные примеры классов-хранилищ данных. До сих пор все приводимые примеры хранилищ могли содержать данные только одного базового типа.

```
class Stack
{
  private:
    int st[MAX];        // целочисленный массив
    int top;            // Индекс вершины стека
  public:
    Stack();            // конструктор
    void push(int var); // int-аргумент
    int pop();          // возвращает значение int
};
```

Если бы мы захотели хранить значения типа long, пришлось бы написать абсолютно новое определение класса:

```
class LongStack
{
  private:
    long st[MAX];       // массив long
    int top;            // Индекс вершины стека
  public:
    LongStack0;         // конструктор
    void push(long var); // int-аргумент
    long pop();         // возвращает значение int
};
```

Мотивация использования

- Подобным же образом нам пришлось бы создавать классы для хранения данных каждого типа, если бы не шаблоны классов, которые и тут приходят к нам на помощь. С их помощью можно написать такую спецификацию класса, которая сохранит все, что попросят, без лишних вопросов. Создадим вариацию на тему **stack** для демонстрации возможностей шаблонов классов.

Реализация шаблона класса Stack

```cpp
// tempstak.cpp
// реализация стека в виде шаблона
#include <iostream>
using namespace std;
const int MAX = 100;
//////////////////////////////////////////////////////////////
template <class Type>
class Stack
{
private:
    Type st[MAX]; // стек: массив любого типа
    int top; // индекс вершины стека
public:
    Stack() // конструктор
    {
        top = -1;
    }
    void push(Type var) // занести число в стек
    {
        st[++top] = var;
    }
    Type pop() // вынуть число из стека
    {
        return st[top--];
    }
};
```

```cpp
int main()
{
    Stack<float> s1; // s1 - объект класса Stack<float>
    s1.push(1111.1F); // занести 3 значения float,
    s1.push(2222.2F); // вытолкнуть 3 значения float
    s1.push(3333.3F);
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
    cout << "3: " << s1.pop() << endl;
    Stack<long> s2; // s2 - объект класса Stack<long>
    s2.push(123123123L); // занести 3 long, вытолкнуть 3 long
    s2.push(234234234L);
    s2.push(345345345L);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
    cout << "3: " << s2.pop() << endl;
    return 0;
}
```

Stack шаблон

Здесь Stack является шаблонным классом. Идея шаблонных классов во многом сходна с идеей шаблонных функций. Ключевые слова template и class Stack говорят о том, что весь класс будет шаблонным

```
template <class Type>
class Stack
  {
  // Данные и методы используют шаблонный аргумент Type
  };
```

Stack s1;

Такое выражение создаст объект s1. Это стек, в котором хранятся числа типа float. Компилятор резервирует область памяти для данных этого объекта, используя тип float везде, где появляется аргумент Type в спецификации класса. Резервируется место и под методы класса (если они не были помещены в память другим объектом типа Stack). Конечно же, методы используют только тип float.

Обработка исключений

- Исключения — механизм обработки ошибок, возникающих во время выполнения программы
- Основные ключевые слова: try, throw, catch
- Подробно механизм исключений рассматривается в следующей лекции

Выводы

1. Шаблон позволяет написать функцию или класс один раз и использовать их с разными типами данных.
2. Объявление начинается с template <class T>; T — параметр-тип, заменяемый компилятором.
3. Код для конкретного типа генерируется только при вызове функции или создании объекта (инстанцирование).
4. Шаблоны классов применяют для контейнеров — стеков, списков, очередей.
5. Объект шаблонного класса создаётся с указанием типа: Stack<float> s1.

Контрольные вопросы

1. Какую проблему решают шаблоны функций?
2. Как записывается шаблон функции? Что означает template <class T>?
3. Когда компилятор генерирует код по шаблону?
4. Что такое шаблон класса? Приведите пример.
5. Как создать объект шаблонного класса Stack для хранения чисел float?
6. Чем шаблоны отличаются от перегрузки функций?

Список литературы

Основная литература

1. Лафоре Р. Объектно-ориентированное программирование в C++. 4-е изд. — СПб.: Питер, 2004.
2. Страуструп Б. Язык программирования C++. Специальное издание. — М.: Бином, 2011.
3. Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Базовый курс. 5-е изд. — М.: Вильямс, 2014.

Дополнительная литература

4. Джосаттис Н. Стандартная библиотека C++: справочное руководство. 2-е изд. — М.: Вильямс, 2014.
5. Майерс С. Эффективный и современный C++. — М.: Вильямс, 2016.
6. Шилдт Г. Полный справочник по C++. 4-е изд. — М.: Вильямс, 2006.

Электронные ресурсы

7. Справочник по C++ и стандартной библиотеке: en.cppreference.com
8. Учебник по C++ (learncpp): www.learncpp.com

- Спасибо за внимание!