Object-oriented programming

Week 7:
Template and metaprogramming

6/2014

What will be discussed?

- Introduction
- Function templates
- Class templates
- Static class members
- Metaprogramming

Introduction

- Frequently, we have to implement the same functions or classes for arguments on different data types

- The templates enable us to implement the function only once to be used for different argument data types

An example: function template

The same function for different data types:

```
int bigger(int a, int b)
{
   return (a>b) ? a : b;
}
```

```
float bigger(float a, float b)
{
   return (a>b) ? a : b;
}
```

Templates enable us to write the function once:

```
TYPE bigger(TYPE a, TYPE b)
{
   return (a>b) ? a : b;
}
```

Templates

Name of parameters
representing data types

```
template <class TYPE>
TYPE bigger(TYPE a, TYPE b)
{
    return (a>b) ? a : b;
}
```

TYPE is chosen by users and is not a keyword.

The operator > must be defined for **TYPE**

Function template

```
template <class TYPE>
TYPE bigger(TYPE a, TYPE b)
{
    return (a>b) ? a : b;
}
int main()
{
    int x=30, y=50;
    Fraction a(2, 3), b(7, 10);
    cout << "The bigger integer " << bigger(x, y);
    cout << "The bigger fraction " << bigger(a, b);
}
```

template instantiation for given data type

Quiz

- Write the template function for the **`swap`** of 2 numbers.

```
template <class TYPE>
void swap(TYPE& a, TYPE& b)
{
    TYPE temp;
    temp = a;
    a = b;
    b = temp;
}
```

Quiz

- Using templates to implement the **`sort()`** function for any data type
- You can implement any sort algorithm that you are familiar with.

Template: how does it work?

Consider `TYPE bigger(TYPE a, TYPE b)`

- If we invoke the **`bigger`** function for 2 variables of **`int`**. Then:
  - Compiler generate a function, e.g. **`bigger_int`** and replace all **`TYPE`** in the function by **`int`**
  - When the program runs the **`bigger_int`** will run instead of the generic **`bigger`**
- Usually the entire template code is located in a header file

Class template

```
template <class TYPE>
class MyArr
{
   public:
      MyArr();
      MyArr(unsigned);
      ~MyArr();
      TYPE& operator[](unsigned);
      const TYPE& operator[](unsigned) const;
      MyArr<TYPE>& operator=(const MyArr<TYPE>&);
   private:
      TYPE* pArr;
      unsigned size;
};
```

Class template

- We can use template parameters as data types in class. In the last example, we have
  - TYPE
  - MyArr<TYPE>
- They are used as normal data types for variable declaration, function arguments and function return… in the class

Template instantiation

- From the last example
  - **`MyArr`**: is just a name of the template class
  - **`MyArr<float>:`** name of the class, an array of float
- Declaring objects
  - **`MyArr<Fraction> arrFrac(200);`**

➔ Compiler will generate an actual class based on the generic template class

Template class definition

At the point of declaration

```
template <class T>
class MyClass
{
public:
   MyClass() {...}
};
```

At the point of definition

```
template <class T>
MyClass<T>::MyClass()
{
    ...
}
```

Template params are not data type

- Besides the data type params, class can have numbers as params:

```
template <class TYPE, int size>
class List
{
   public:
      ...
   private:
      TYPE arr[size];
};
```

An example

- Then, the list with 100 elements in the example below will be generated at compile time:


```
int main()
{
   List<int, 100> a;
   return 0;
}
```

Static class members

- Static class members are common (i.e. use the same memory slot) for all instances of that class.

- Static member functions don’t have **`this`** pointer and are only able to use static members of that class.
- You can refer to a static member by direct access (i.e. dot) via any object of that class or by scope resolution via class name.

Static class members

- Static class members have to be defined exactly once in the program.

```
class A
{
      static int x;
      int z;
public:
      static void doSomething();
      int test() { return x+z; }
};
int A::x=5;
```

Metaprogramming

- Templates enable C++ compiler work as an interpreter. Programs are interpreted at the compiling time.
- For example, loops or conditional checking can be replaced by recursive templates

An example

```
template<int N>
class Factorial
{
public:
   static const int val = N*Factorial<N-1>::val;
};
template<>
class Factorial<0> {
public:
      static const int val=1;
};
int main()
{
   cout << Factorial<5>::val << endl; return 0;
}
```

Metaprogramming

- Those programs have results at the compiling time.
- This technique becomes very useful and powerful in C++ programming.
- See **C++ Template Metaprogramming** book by David Abrahams for further information on this

Exercise

- Using **metaprogramming** to write the **`bubblesort`** for an array of integers