Object-Oriented Programming

Lesson 12

Templates
Nested classes
Iterators

Templates

Inheritance and composition provide a way to reuse object code.

The template feature in C++ provides a way to reuse source code.

The template keyword tells the compiler that the code that follows will manipulate one or more unspecified types.

At the time the actual code is generated from the template, those types must be specified so that the compiler can substitute them and generate a new source code from the template code.

Function templates

A function template creates new functions based on type parameters. For example, you can define a function template which sorts an array of objects of any type.

The compiler will generate automatically functions required for sorting objects of particular types, such as int, float, Figure or any other.

The function template definition starts with the keyword template followed by a list of parameters enclosed in < and >. Each parameter is preceded with the keyword class:

```
template <class M>
template <class M, class C>
```

Function templates

Consider the example of a function template to find the largest value in an array of elements. The function template, *find_max*, declares a single parameter T (actually any valid identifier can be used, here T was chosen), which defines the type of the array elements to be analysed by *find_max* function.

```
template <class T>
T find_max (T* array, int size)
{  assert (size > 0);
   T max = array[0];
   for (int i = 1; i < size; i++)
      if (array[i] > max)
         max = array[i];
   return max;
}
```

When the compiler detects a call of *find_max* function, the type T will be replaced by the type specified in the invocation of the function. As a result, a new function will be created by the compiler, which can process an array of the specified type.

```
int array_int[] = { 1, 45, 23, 9, 5};
char array_char[] = { 'd', 'a', 'b', 'Z' };

cout << find_max<int>(array_int, 5);
cout << find_max<char>(array_char, 5);
```

Class templates

When applied to a class, the template keyword tells the compiler that the class definition that follows will manipulate one or more unspecified types.

At the time the actual class code is generated from the template, those types must be specified so that the compiler can substitute them.

Class templates are typically used to implement containers (which hold (pointers to) objects).

Class templates

Array of integers:

```
class CArray
{
   unsigned m_size;
   int* m_array;
public:
   CArray(unsigned size)
   { m_array = new int[m

   ~CArray() { delete []

   int& operator[] (unsi
   { assert (indice < m_
     return m_array[indi
};
```

Array of doubles:

```
class CArray
{
   unsigned m_size;
   double* m_array;
public:
   CArray(unsigned size) : m_size(size)
   { m_array = new double[m_size]; }

   ~CArray() { delete [] m_array; }

   double& operator[] (unsigned indice)
   { assert (indice < m_size);
     return m_array[indice]; }
};
```

Array of ...

Templates

Array of any type:

```
template <class T>
class TArray
{
    unsigned m_size;
    T* m_array;
public:
    TArray(unsigned size
    { m_array = new T[m_

    ~TArray() { delete [

    T& operator[] (unsig
    { assert (indice < m
      return m_array[ind
};
```

Instantiation of the array of integers

```
void main()
{
  TArray<int> a(5);
  for (int i = 0; i < 5; i++)
  {
    a[i] = i*i;
    cout << a[i] << endl;
  }

  TArray<float> b(5);
  for (int i = 0; i < 5; i++)
  {
    b[i] = static_cast<float>(i*i)/10;
    cout << b[i] << endl;
  }
}
```

Instantiation of the array of floats

Templates

Non-inline function definitions:

```
template <class T>
class TArray
{  //...
   T& operator[] (unsigned indice);


};
```

Any reference to a template's class name must be accompanied by its template argument list, as in **TArray<T>::operator[ ]**.

```
template <class T>
T& TArray<T>::operator[] (unsigned indice)
{
      assert (indice < m_size);
      return m_array[indice];
}
```

Templates

Even if you create non-inline function definitions, you'll usually want to put all declarations and definitions for a template into a header file.

Anything preceded by **template<...>** means the compiler won't allocate storage for it at that point, but will instead wait until it's told to (by a template instantiation), and that somewhere in the compiler and linker there's a mechanism for removing multiple definitions of an identical template.

Template arguments are not limited to built-in types:

```
TArray<CMyClass> a(5);
```

Nested classes

Nested classes are defined within other classes.

```
class One
{
    int i;
    class Other //nested class
    {
        int j;
    public:
         Other (int a);
    };
};

One::Other::Other(int a) : j(a)
{ }
```

A nested class, by default, only has access to public data members of the enclosing class.

The class `Other` is not visible ouside the scope of the class `One`.
Objects of the class `One` **do not** include any sub-objects of type `Other`.

Classes aninhadas

Making a class nested doesn't automatically give it access to **private** members. To accomplish this, you must follow a particular form:

1) Declare (without defining) the nested class;
2) Declare it as a **friend**;
3) Define the class.

```
class One
{
   int i;
   class Other;
   friend class Other;
   class Other //nested class
   {
       int j;
   public:
        Other (int a);
   };
};
```

Iterators

Iterator is an object that moves through a collection/container of other objects and selects them one at a time, without providing direct access to the implementation of the container.
It's more common to see an "iterator" class nested within the class that it services.

```
template <class T> class TArray
{       T** m_array;   unsigned m_nSize;
public:
        class iterator;
        friend class iterator;
        class iterator
        {       const TArray& t;       unsigned index;
        public:
                iterator(const TArray<T>& a, unsigned s = 0);
        };
        const iterator begin() const;
        const iterator end() const;
};
```

```
TArray<CFigure>::iterator iter = ar_fig.begin();
while (iter != ar_fig.end())
        cout << **iter++ << " ";
```

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Chapters 5, 12, 16