Templates and Smart Pointers

Chapter 12

Objectives

- **Describe the C++ *template* mechanism and implement programs using templates.**
- **Understand how to write simple template functions and classes.**
- **Understand the principles behind generic programming.**
- **Implement a general array class in C++ using templates.**
- **Understand the basic elements of the Standard Template Library.**
- **Understand how to utilize smart pointers.**
- **Gain experience through code walk-throughs and lab exercises.**
    - The example programs are in the chapter directory.
    - Labs located in Labs/Lab12

Macros

- **The first versions of C++ required you to use the C preprocessor macro facility to avoid repetitive coding:**
  - #define Max(a, b) (a > b) ? a : b

- **But macros are not type safe, and you can get anomalous results from the blind text substitution.**
  - Max(x++, y++);
    - – Either **x** or **y** winds up getting incremented twice, depending on which is larger.

Macros (continued)

- **The preprocessor knows nothing about the C++ language, it just does text replacement.**
- **Examine the sample program in the [Macromax folder.]**

General Purpose Functions

- **Consider defining a general purpose function *Max* that will find that maximum of two quantities for which a greater than operator (>) is defined.**
    - The code snippets on the next slide differ only in the type being compared.

General Purpose Functions Example

```
int Max(const int& a, const int& b){
    if (a > b) return a;
    else return b;
}
```

```
double Max(const double& a, const double& b){
    if (a > b) return a;
    else return b;
}
```

Function Templates

- A *function template* allows you to specify a set of functions that are based on the same code but act on different types.

```
template<typename T>
T Max(const T& a, const T& b){
        return (a > b) ? a : b;
}
```

- **A class type can be used as a parameter by using the expression class T, or typename T, where T is a dummy parameter.**
- **Use T (or any parameter name) in the function definition wherever a type identifier would be used.**

Call a template function

- **When a call is encountered (either explicitly or when the address of a function is used), the compiler will instantiate a version of the function specialized for the actual type used.**

```
int a; int b = 10; int c = 5;
a = Max(b, c);
double d; double e = 10.5; double f = 23.2;
d = Max(e, f);
long (*pfunc)(long, long) = Max;
long g = pfunc(4, 5);
```

- Identical instantiations, even in different modules, will be optimized out of the final executable.

Template Parameters

- **Template function definitions can have multiple parameters.**
  - template <class T, class U> int Foo(...) {...}
- **A compilation error will be generated if the template parameters cannot be matched properly.**

```
long a = 23; long b = 10; double c = 5.3;
a = Max(b, c);        // compilation error
c = Max(a, b);        // ok

void func(long (*)(long, long)){}
void func(double(*)(double, double)){}
// compilation error, which version of func?
func(Max);
```

Template Parameter Conversion

- **The compiler will only use an exact match for the template parameters.**
  - No type conversions will be attempted for matching arguments in a function template.
  - No promotions are attempted (i.e. **unsigned int** does not match **int**)
  - No user defined conversions are attempted.

Function Template Problem

- **Given the function template Max defined earlier, what is the result of the following code fragment?**

```
const char* a = "rook"; const char* b = "checkmate";
const char* c = Max(a, b);
```

Function Template Problem

- **The compiler will generate the following code:**

```
const char* Max(const char* a, const char* b) {
    if (a > b) return a;
    else return b;
}
```

The addresses are compared, not the string values!

Function Template Problem (continued)

- **It is legal to declare an ordinary function version of *Max()* to be used in place of the template definition**

```
const char* Max(const char* a, const char* b){
   if (strcmp(a, b) > 0) return a;
   else return b;
}
```

Function Template Demo

- **Examine the sample program in the TemplateMax folder. Add code to call one of the provided Compare functions.**

Generic Programming with Function Objects

- **Function objects behave like functions**
  - **overload operator()**
- **Template code often parameterized by function objects**
  - **apply generic operation to elements**
- **The use of function templates allow methods to be passed as parameters**
  - **apply generic operation to elements**
  - **compile-time polymorphism**
- **Review the sample programs in the TemplateOperate and FunctionObject folders.**

Class Templates

- **A class template definition enables you to define a generic class.**
  - Class could store any type

```
template <typename E> class Array{
public:
  Array(int size = 10);
  ~Array();
  Array& operator=(Array&);
  long size() const;
  E& operator[]() const;
private:
  long m_lSize;
  E* m_array;
};
```

Class Templates (continued)

- **The *template<argument-list>* expression is also used in front of the code implementation (*also in .h file*).**
- **If you try to keep the code implementation in a separate .cpp file, you will get linker errors, because the code file that uses the template does not have access to the template code.**

```
template <typename  E>
long Array<E>::size() const {
   return m_lSize;
}
```

Class Template Instantiation

- **There is no relationship between instances of a template class.**
  - **Given two instances of *Foo<T>, Foo<int>* and *Foo<double>,* the compiler generates two completely separate classes.**
- **Unlike a function template, you cannot deduce the arguments for a class template from its context.**
- **The compiler will generate code only for those methods of a class template that are actually used.**
  - **A compilation error in an unused class template method will not be detected until the method is used.**

Class Template Instantiation (continued)

- **When the class is instantiated, the class name is used, followed by *actual* arguments between angle brackets.**

```
Array<int> array1(7);      // array of 7  integers
Array<string> array2(5);   // array of 5  strings
Array<double> array3;      // array of 10 doubles
Array<double> array4(5);   // array of 5  doubles
```

The compiler generates three copies of the **`Array`** code (not four since only one copy is needed for **`double`**)

Class Template Demos

- **Examine and run the sample programs in the folders [TemplateArray] and [TemplateSet]**

Standard Template Library

- **The Standard Template Library (STL) is a general purpose library that contains data structures and generic algorithms.**
  - STL is a part of the ANSI C++ standard.
- **The key idea is that algorithms can be written in a fashion that is independent of the data structure they operate on.**
  - Templates make it possible to write algorithms with generic types.
  - The algorithms are written in terms of template classes called iterators operating on generic types.
  - The iterators are implemented by the data structure (container) classes so they understand how to navigate a particular container.
- **Since STL is based on templates, there are no libraries to link, you just have to include the proper files.**
  - The ANSI include files, including the STL, do not have the .h suffix.

STL Components

- ***Containers* are abstract data structures.**
  - Sequence containers: C++ arrays, vectors, deques, lists
  - Sorted associative containers: sets, multisets, maps, etc.
  - Include files: <vector>, <list>, <set>, <map>
- ***Iterators* are used to move through items in a container. They have the semantics of a pointer.**
  - Implement operators **==**, !=, *, ++
  - May implement operators --, +=, -=, +, -, <, >, <=, >=
  - Include file: <iterator>
- ***Function Objects* encapsulate a function for use by other STL components**
  - overloads the function call operator: *operator()*
- ***Adapters* map a new interface to an existing container, iterator, or function object.**
  - Include files: <stack>, <queue>, etc.
- ***Algorithms* perform common operations, such as sorting, on containers.**
  - Include files: <algorithm>, <numeric>
- ***Allocators* are classes that handle the details of memory management.**

In-class discussion / work-along

- **Following is a general approach (non-template) to locating a number in an array of ints:**

```
int* find(int *begin, int *end, int value) {
        int *pCurrent = begin;
        while (pCurrent != end && *pCurrent != value) { ++pCurrent; }
        return pCurrent;
}
```

- **How could this be generalized using templates?**
- **How would you implement a generalized find_if?**
- **How would you implement a generalized sort?**

STL Elements of a find (1/3)

- **STL *find* from <algorithm>:**

```
template< typename _II, typename _Ty>
inline _II find(_II _F, _II _L, const _Ty& _V)
{
    for (; _F != _L; ++_F)
        if (*_F == _V)
                break;
    return (_F);
}
```

STL Elements of a find (2/3)

- **This algorithm searches a range for a particular value.**
  - The value _V is a reference of type _Ty
  - The start of the range is at position _F and the end of the range is _L. These positions are represented by iterators of type _II.
  - Since the algorithm uses **==,** !=, ++, and * operators those are the only operators that the iterator has to overload.

STL Elements of a find (3/3)

- **The iterator is implemented by the container (as an embedded class) so that it knows how to navigate the container.**
- Here a list is used as a container whose elements are of a single type:

```
list<int>::iterator start = lst1.begin();
list<int>::iterator finish = lst1.end();
list<int>::iterator where;
where = find(start, finish, 5);
```

STL demos

- **Lists are sequence containers insert and erase operations anywhere within the sequence, and iteration in both directions.**
  - Examine and run the program in the folder List
- **Vectors are sequence containers representing arrays that can change in size.**
  - Examine and run the program in the folder Vector
- **The map container is a sorted associative container. It relates unique keys of a given type to a value.**
  - Examine and run the program in the folder Map.

Smart Pointers

- **Smart pointers are objects that behave like pointers, but with added features such as automatic memory management, increased code safety, and flexibility.**
- **Smart pointers come in a variety of types, each with its own unique features and benefits.**
  - **The shared_ptr type supports shared ownership.**
    - Counts the number of owners.
    - When count is zero (all owners have released ownership), the object is deleted.
  - **The unique_ptr maintains a unique instance of an object via a pointer.**
    - No reference counting.
    - When moved, the original pointer is set to null.
    - Copy not allowed
  - **The weak_ptr type refer to a weak reference to memory.**
    - Weak pointers create a shared reference without adding to the reference count.
    - Create a weak pointer to optionally preserve a pointer in memory

unique_ptr

- **When you need a smart pointer for a plain C++ object, use unique_ptr**
- **Unlike share_ptr, a unique_ptr does not share its pointer.**
- **Syntax:**

```
unique_ptr<double> sp1(new double(100));
        or
unique_ptr<double> sp2 = std::make_unique<double>(5);  /* introduced in C++ 14 */
```

shared_ptr

- **Use a shared_ptr when more than one owner might have to manage the lifetime of the object in memory.**
- **After you initialize a shared_ptr you can copy it, pass it by value in function arguments, and assign it to other shared_ptr instances**
- **Syntax:**

```
shared_ptr<double> sp1(new double(100));
        or
shared_ptr<double> sp2 = std::make_shared<double>(5);
```

weak_ptr

- **Provides a way to access the underlying object of a shared_ptr without causing the reference count to be incremented.**
  - **Typically, this need arises when you have cyclic references between shared_ptr instances.**
  - **By using a weak_ptr, you can create a shared_ptr that joins to an existing set of related instances, but only if the underlying memory resource is still valid.**
- **Syntax:**

```
shared_ptr<double> sp(new double(100));
weak_ptr<double> wp(sp);
```

Additional Features

- **Both unique_ptr and shared_ptr allow you to specify a custom deleter function or function object that will be called when the pointer is deleted.**
- **shared_ptr allows you to specify a custom allocator object that will be used to allocate memory for the reference count and control block associated with the pointer.**
- **With a shared_ptr you can specify a custom hash function.**
  - **This is useful as a key in an unordered container like unordered_map.**
- **If you're comparing smart pointers with == or != operators, you can specify a custom comparison operator that will be used to compare the underlying raw pointers.**

Summary

- **Templates provide a mechanism that is part of the C++ language that enables one to program in a generic, type independent, fashion.**
- **Most programmers will use templates written by others, rather than write their own templates. Nonetheless, understanding how templates work is important to the correct use of templates.**
- **The key idea in generic programming is to separate the algorithm's logic from the data structure.**
  - A data structure called an iterator allows an algorithm to navigate through a data structure without knowing how the data structure is organized
  - This concept is the foundation for the Standard Template Library (STL) which is now part of the ANSI C++ Library.