Creative Software Programming

12 – Template

Today's Topics

- Intro to Generic Programming
- Function Template
- Class Template
- Review Standard Template Library (STL)
  - A set of C++ template classes
- Templates and Inheritance

C++ Template

- A C++ feature that allows functions and classes to operate with ***generic types***.
  - You can think of *generic type* as *to-be-specified-later type.*

- This allows a function or class to **work on many different data types without being rewritten** for each one.[wikipedia]

- The C++ Standard Template Library (STL) provides many useful functions within a framework of connected **templates**.

Generic Programming

- Style of computer programming
  - Algorithms are written in terms of ***to-be-specified-later*** **types** that are then instantiated when needed for specific types provided as parameters.[wikipedia]
  - C++ Standard Template Library (STL).
    - Best known example
    - Data containers such as vector, list, map, etc.
    - Algorithms such as sorting, searching, hashing, etc.

Direct Approach

- Need ***K*** sorting algorithms to handle ***K*** different data types

```
// Suppose we want to sort
// an integer array.

void SelectionSort(int* array,
                   int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1;
         j < size; ++j) {
      if (array[min_idx] > array[j])
{
        min_idx = j;
      }
    }
    // Swap array[i] and
    // array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
// We also want to sort
// a double array.

void SelectionSort(double* array,
                   int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1;
         j < size; ++j) {
      if (array[min_idx] > array[j])
{
        min_idx = j;
      }
    }
    // Swap array[i] and
    // array[min_idx].
    double tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

Generic Approach

- C++ template allows us to avoid this repeated codes.
- ***Functions*** and ***classes*** can be ***templated***.

```
// Suppose we want to sort an array of type T.

template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j]) {
        min_idx = j;
      }
    }
    // Swap array[i] and array[min_idx].
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

Function Template

- A generic function description
  - defines a ***function*** in terms of a ***generic type***
    - A specific type, such as *int* or *double*, can be substituted.
- Passing a specific type as a parameter to a template
  - Compiler generates a function for that particular type
- Write functions of the same algorithm once for various types.

Function Template: Basics

- Example
  - Swap function

```
template <typename T>
// naming the arbitrary type T. Programmers use simple names such as T .
void Swap(T &a, T &b) {
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

  - **The template does not create any functions**
    - Let the compiler know how to define a function

Function Template : Example

```
template <typename T>
void Swap(T &a, T &b) {
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

```
Output:
i, j = 10, 20
template int swapper:
Now i, j = 20, 10
x, y = 24.5, 81.7
template double
swapper:
Now x, y = 81.7, 24.5
```

```
template <typename T> // or class T
void Swap(T &a, T &b);

int main() {
  int i = 10;
  int j = 20;
  cout << "i, j = " << i << ", " << j << endl;
  cout << "template int swapper:\n";
  Swap<int>(i,j);
  // generates void Swap(int &, int &)
  cout << "Now i, j = " << i << ", " << j <<
endl;

  double x = 24.5;
  double y = 81.7;
  cout << "x, y = " << x << ", " << y << endl;
  cout << "template double swapper:\n";
  Swap<double>(x,y);
  // generates void Swap(double &, double &)
  cout << "Now x, y = " << x << ", " << y <<
endl;
  return 0;
}
```

Function Template : Example

- Templates are "instantiated" at compile time.
- "Function template instance"

```cpp
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

Compiler internally generates and adds below code

```cpp
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```cpp
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Template Argument Deduction

- You can **omit** any template argument that the compiler can deduce by the usage and context of that template function call.

```
int i = 10;
int j = 20;
Swap<int>(i, j);
```

=

```
int i = 10;
int j = 20;
Swap(i, j);
```

Function Template : Overloading

- Overloading template functions

```cpp
template <typename T>
void Swap(T &a, T &b) {
  T temp;
  temp = a;
  a = b;
  b = temp;
}

template <typename T>
void Swap(T* a, T* b, int n) {
  T temp;
  for (int i = 0; i < n; i++)
{
    temp = a[i];
    a[i] = b[i];
    b[i] = temp;
  };
}
```

```cpp
int main() {
  int i = 10, j = 20;
  cout << "i, j = " << i << ", " << j <<
endl;
  cout << "Swap scalars" << endl;
  Swap(i,j);  // generates Swap(int &, int &)
  cout << "i, j = " << i << ", " << j <<
endl;
  cout << "*******************" << endl;
  int d1[] = {1,2};
  int d2[] = {3,4};
  int n = 2;
  cout << "d1[0]=" << d1[0]
      << ", d1[1]=" << d1[1] << endl;
  cout << "d2[0]=" << d2[0]
      << ", d2[1]=" << d2[1] << endl;
  cout << "Swap arrays" << endl;
  Swap(d1,d2, n);
  // generates void Swap(int *, int *, int)
  cout << "d1[0]=" << d1[0]
      << ", d1[1]=" << d1[1] << endl;
  cout << "d2[0]=" << d2[0]
      << ", d2[1]=" << d2[1] << endl;
  return 0;
}
```

Function Template : Overloading

- Overloading template functions; result

```
Output:

i, j = 10, 20
Swap scalars
i, j = 20, 10
*******************
d1[0]=1, d1[1]=2
d2[0]=3, d2[1]=4
Swap arrays
d1[0]=3, d1[1]=4
d2[0]=1, d2[1]=2
```

Quiz #1

```
#include <iostream>
using namespace std;

template <typename T>
T MyMax(T x, T y) {
   return (x > y)? x: y;
}

int main() {
  cout << MyMax<int>(1, 2) << endl;
  cout << MyMax(3.1, 7.5) << endl;
  cout << MyMax('g', 'e') << endl;

  return 0;
}
```

- What is the expected output of this program? (If a compile error is expected, just write down "error").

Class Template

- Class members can be templated
  - Define a class in a generic fashion (type-independent)
  - Allow to reuse code
    - Inheritance & containment aren't always the solution

```
class Stack1 {
 private:
  enum { MAX = 10 };
  // constant specific to class
  Item1 items[MAX];
  // holds stack items
  int top; // index for top stack item

 public:
  Stack1();
};
```

```
class Stack2 {
 private:
  enum { MAX = 10 };
  // constant specific to class
  Item2 items[MAX];
  // holds stack items
  int top; // index for top stack item

 public:
  Stack2();
};
```

Class Template: Basic

- How to use:

```
template <typename T>
// let the compiler know that you're about to define a template
class Stack {
 private:
  enum { MAX = 10 };  // constant specific to class
  T items[MAX];  // holds stack items (type-independent)
  int top;  // index for top stack item

 public:
  Stack();
};
```

  - When a template is invoked, ***T*** will be replaced with a specific type
    - E.g., *int* or *string*
  - ***Generic type name***, ***T***, to identify the type to be stored in the stack

Class Template: Example

```
template <typename T>
class MyPair {
  T a, b;
 public:
  MyPair(T first, T second) {
    a = first, b = second;
  }
  T get_max();
};

template <typename T>
T MyPair<T>::get_max() {
  T retval;
  retval = a > b? a : b;
  return retval;
}
```

```
int main() {
  int a_i = 100, b_i = 75;
  MyPair<int> my_pair_i(a_i, b_i);
  cout << "max(" << a_i << "," << b_i
      << ")=" << my_pair_i.get_max() << endl;

  double a_d = 1.5, b_d = -3.5;
  MyPair<double> my_pair_d(a_d, b_d);
  cout << "max(" << a_d << "," << b_d
      << ")=" << my_pair_d.get_max() << endl;

  return 0;
}
```

Output:

```
max(100,75)=100
max(1.5,-3.5)=1.5
```

Class Template: Example

- Templates are "instantiated" at compile time.
- "Class template instance"

```
MyPair<int> my_i(a_i, b_i);
```

```
class MyPair {
  int a, b;
 public:
  MyPair(int first, int second) {
    a = first, b = second;
  }
  int get_max();
};

int MyPair<int>::get_max() {
  int retval;
  retval = a > b? a : b;
  return retval;
}
```

Class Template: Closer Look at

- Types for the **MyPair** <T>
  - Both built-in types and classes are allowed
  - How about pointers?
    - Won't work very well without major modifications
    - Need to take care

```
int main() {
  int a_i = 100, b_i = 75;
  MyPair<int*> myobject_i(&a_i, &b_i);
  cout << "max(" << a_i << "," << b_i << ")="
       << myobject_i.get_max() << endl;
  return 0;
}
```

```
Output:

max(100,75)=0x22fe2c
```

Member Function Template

- Can be used to provide additional template parameters other than those of the class template.

```
template<typename T>
class X {
 public:
   template<typename U>
   void mf(const U& u);
};

template<typename T>
template <typename U>
void X<T>::mf(const U& u) {
  ...
}

int main() {
  ...
}
```

typename & class keyword

- ‘typename’ can always be replaced by keyword ‘class’.

```
template <class First, class Second>
// Same as <typename First, typename Second>.
struct Pair {
  First first;
  Second second;
  Pair(const First& f, const Second& s) : first(f), second(s) {}
};

template <class First, class Second>
Pair<First, Second> MakePair(const First& first,
                             const Second& second) {
   return Pair<First, Second>(first, second);
}
```

```
int main (){
  Pair<int, int> p = MakePair(10, 10);
  // == MakePair<int, int>(10, 10);
  Pair<int, int> q = Pair<int, int>(20, 20);
  return 0;
}
```

Non-type Template Parameter

- A non-type template parameter is...
  - provided within a template argument list
  - **an expression whose value can be determined at compile time**
    - ***constant expressions***
  - treated as **const**
  - e.g.,

```
template<class T, int size>
```

Non-type Template Parameter

```
template<class T, int size>
class MyFilebuf {
  T* filepos;
  int array[size];

 public:
  MyFilebuf() { /* ... */ }
  ~MyFilebuf() {}
  ...
};
```

```
int main (){
  MyFilebuf<double, 200> x; // create object x of class
  MyFilebuf<double, 200.0> y;
  // error, 200.0 is a double, not an int
  return 0;
}
```

Non-type Template Parameter

```
template <int i>
class C {
 public:
  int array[i];
  int k;

  C() { k = i; }
};
```

```
int main() {
  C<100> a; // can be instantiated
  C<200> b; // can be instantiated
  return 0;
}
```

Quiz #2

```
template<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>
class Matrix {
 private:
  Scalar _rawdata[RowsAtCompileTime][ColsAtCompileTime];

 public:
  // ...
};

int main () {
  __________(a)___________;
  return 0;
}
```

- Using the Matrix class template above, you want to create a 3 x 3 matrix object named `mat` which has elements `double` type. Write the code for this in (a). (Use default constructor)

STL Revisit

- STL defines powerful, **template-based**, reusable components
- STL uses **template-based genetic programming**
- **A collection of useful templates** for handling various kinds of data structure and algorithms **with generic types**
  - Containers
    - Data structures that store objects of any type
  - Iterators
    - Used to manipulate container elements
  - Algorithms
    - Operations on containers for searching, sorting and many others

Containers Revisit

- Sequence: contiguous blocks of objects
  - Vectors: insertion at end, random access
  - List: insertion anywhere, sequential access
  - Deque (double-ended queue): insertion at either end, random access
- Container adapter
  - Stack: Last In Last Out
  - queue: First In First Out
- Associative container: a generalization of sequence
  - Indexed by any type (vs. sequences are indexed by integers)
  - Set: add or delete elements, query for membership…
  - Map: a mapping from one type (key) to another type (value)
  - Multimaps: maps that associate a key with several values

vector - a resizable array

```
#include <iostream>
#include <vector>
using namespace std;

int main(void){

    vector<int> intVec(10);

    for(int i=0; i< 10; i++){
            cout << "input!";
            cin >> intVec[i];
    }

     for(int i=0; i< 10; i++){
            cout << intVec[i] << " " ;
    }
    cout << endl;
    return 0;
}
```

STL: vector

- Standard library header <vector>
  - A class template
  - Templated member functions/variables

```
template <class T, class Allocator = allocator<T> >
class vector {
 public:
  // types:
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef T value_type;
  typedef Allocator allocator_type;
  typedef typename allocator_traits<Allocator>::pointer pointer;
  typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
};
```

STL: vector

- Standard library header <vector>
  - Constructors/destructor

```
template <class T, class Allocator = allocator<T> >
class vector {
 public:
  // construct/copy/destroy:
  explicit vector(const Allocator& = Allocator());
  explicit vector(size_type n);
  vector(size_type n, const T& value,const Allocator& = Allocator());
  template <class InputIterator>
  vector(InputIterator first, InputIterator last,
         const Allocator& = Allocator());
  vector(const vector<T,Allocator>& x);
  vector(vector&&);
  vector(const vector&, const Allocator&);
  vector(vector&&, const Allocator&);
  vector(initializer_list<T>, const Allocator& = Allocator());
  ~vector();
};
```

STL: vector

- Standard library header <vector>
  - Assignment operators / member functions

```
template <class T, class Allocator = allocator<T> >
class vector {
 public:
  vector<T, Allocator>& operator=(const vector<T, Allocator>& x);
  vector<T, Allocator>& operator=(vector<T, Allocator>&& x);
  vector& operator=(initializer_list<T>);
  template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
  void assign(size_type n, const T& t);
  void assign(initializer_list<T>);
  allocator_type get_allocator() const noexcept;
};
```

STL: vector

- Standard library header <vector>
  - Iterators
    - begin(), end(), rbegin(), rend(), …
  - Capacity
    - size(), resize(), capacity(), capacity(), empty(), reserve(), …
  - Element access
    - [], at(), front(), back()
  - Modifiers
    - push_back(), pop_back(), insert(), erase(), swap(), clear(), …

  - Everything is templated!!

Class template vs. Template class

- The correct term is "**class template**".
- "template class" does not exist in the C++ standard.
  - E.g., a class template, but not a class

```
template<typename T>
class MyClassTemplate { ... };
```

  - E.g., a class, but not a class template

```
MyClassTemplate<int>
```

Templates and Inheritance

- Derivation works the same as with ordinary classes.
- One can create a new template object from an existing template.

```
template<class T>
class CountedQue : public QueType<T> {
 public:
  CountedQue();
  void Enqueue(T new_item);
  void Dequeue(T& item);
  int Length() const;

 private:
  int length;
};
```

Templates and Inheritance

- Overidding

```
template<class T>
class Base {
 public:
  void set(const T& val) { data = val; }
 private:
  T data;
};

template<class T>
class Derived : public Base<T> {
// should be Base<T>, not just Base
 public:
  void set(const T& val);
};

template<class T>
void Derived<T>::set(const T& v) {
  Base<T>::set(v);
  // should be Base<T>, not just Base
}
```

Templates and Inheritance

- Derived class may have its own template parameters.

```
template<class T>
class Base {
 public:
  void set(const T& val) { data = val; }

 private:
  T data;
};

template<class T, class U>
class Derived : public Base<T> {
// should be Base<T>, not just Base
 public:
  void set(const T& val);

 private:
  U derived_data;
};
```

Templates and Inheritance

- A derived class may inherits from an explicit instance of the base class template.

```
template<class T>
class Base {
 public:
  void set(const T& val) { data = val; }
  T get() { return data; }

 private:
  T data;
};

class Derived : public Base<int> {
  // explicit instance of the base class
 public:
  int get(){ return Base<int>::get(); }
};
```

Templates and Inheritance

- Parameterized inheritance

```
class Shape {
 public:
  void Display() { cout << "show" << endl; }
};

template<class T>
class Rectangle : public T {
  // base class is the template parameter
 public:
  void Display() { T::Display(); }
};

int main() {
  Rectangle<Shape> rect;
};
```

Quiz #3

```cpp
#include <iostream>
using namespace std;

template <typename T>
class Class1 {
  T var1;

 public:
  Class1(const T& v) : var1(v) {}
  void foo() {
    cout << var1 << endl;
  }
};

template <typename T>
class Class2 : public Class1<T> {
  T var2;

 public:
  Class2(const T& v) : Class1(v) {}
  void test() {
    Class1<T>::foo();
  }
};
```

```cpp
int main() {
  Class2<int> class2(10);
  class2.test();

  return 0;
}
```

- What is the expected output of this program? (If a compile error is expected, just write down "error").

Templates and Static Members

- General classes
  - **static** member variables can be shared between all objects
- Classes template instances
  - Each class (e.g., MyTemplate<int>, MyTemplate<double>) has its own copy of **static** member variables
  - Each class template instance gets its own copy of **static** member functions

Templates and Static Members

- Example

```
template <class T>
class TemplatedClass {
 public:
  static T x;
};

template <class T>
T TemplatedClass<T>::x;

int main() {
  TemplatedClass<int>::x = 1;
  cout << TemplatedClass<int>::x << endl;
  cout << TemplatedClass<float>::x << endl;
  return 1;
}
```

```
Output:
1
0
```

Summary of Three Approaches

Naïve Approach	Function Overloading	Template Functions
▪ Different Function Definitions ▪ Different Function Names	▪ Different Function Definitions ▪ Same Function Name	▪ One Function Definition (a function template) ▪ Compiler Generates Individual Functions

Next Time

- Next lecture:
  - 13 - Exception Handling