Premature optimization is the root of all evil.

D. Knuth

On the other hand, we cannot ignore efficiency.

Jon Bentley

9.1 Introduction

One of the most useful kinds of classes is the container class, that is, a class that holds objects of some (other) type. Lists, arrays and sets are container classes. However, container classes have the interesting property that the type of objects they contain is of little interest to the definer of a container class, but of crucial importance to the user of a particular container. Thus we want to have the type of the contained object be an argument to a container class. The definer specifies the container class in terms of that argument, and the users specify what the type of contained object is to be for each particular container (each object of the container class).

In chapter 6, you saw how to create an object of type int_vector, protected_int_vector, range_int_vector and combined_int_vector. If you wanted to build objects of all the above-mentioned classes of type double, we have a problem; int_vector knows only about the integer data type.

One way to solve that problem is to create a copy of all the above four classes and change the data type from int to double and make all the other relevant changes. This solution is not satisfactory. Over a period of time we might require a vector of shapes, vector of complex numbers etc., and it would mean lot of cut and paste.

9.2 Templates

Templates are a mechanism for generating functions and classes based on type parameters, which is why it is also called parameterized types. By using templates, we can design a single class that operates on data of many types, instead of having to create a separate class for each type.

Templates can significantly reduce source code size and increase code flexibility without reducing type safety. They are often used to:

1. Create a type safe collection class (vector, stack etc.) that can operate on data of any type.
2. Add extra type checking for functions that would otherwise take void pointers.

These tasks can be done without using templates; however using templates offer several advantages:

1. Templates are easier to write. You create only one generic version of your class or function instead of manually creating specific classes.
2. Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
3. Templates are type safe. Since the types that templates act upon are known at compile time, the compiler can perform type checking before an error occurs.

We had seen in chapter 2, how we can write a type safe function that returns the minimum of two parameters without using templates, you would have to write a set of overloaded functions like this:

```
// min for ints
int min(int a, int b)
{
    return(a < b) ? a : b;
}

// min for longs
long min(long a, long b)
{
    return(a < b) ? a : b;
}

// min for chars
char min(char a, char b)
{
    return(a < b) ? a : b;
}
```

By using templates, you can reduce this duplication to a single function template:

9.1 Code Listing

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

The template <class T> prefix specifies that a template is being declared and that an argument T of type will be used in the declaration. After its introduction, T is used exactly like other type names. The scope of T extends to the end of the declaration that template <class T> prefixes.

Note that template <class T> says that T is a type name; it need not actually be the name of a class.

The above code defines a family of functions that find the minimum. From this template we can generate functions that will not only find the minimum of two int, long and double types, but also user defined types (complex, string, vector etc.).

In addition, the function template will prevent you from swapping objects of different types, since the compiler knows the types of **a** and **b** parameters at compile time.

Note that all the template parameters inside the angle brackets must be used as parameters for the templated function.

You can call a templated function as you would a non-templated function; no special syntax is needed.

```
int i, j;
double d;

min(i, j);  // Fine
min(i, d);  // Error, different types
```

When a templated function is first called for each type, the compiler creates an "instantiation"; a specialized version of the templated function will be called every time the function is used for the type. If you have several identical instantiations, even in different modules, only one copy of the instantiation will end up in the executable.

Standard type conversions are not applied to templated functions. Instead, the compiler first looks for an "exact match" for the parameters supplied. If this fails, it tries to create a new instantiation to create an "exact match". Finally, the compiler attempts to apply overloading resolutions to find a match for the parameters. If this fails, the compiler generates an error.
In addition, a function template can be overridden, if we need to have a special behavior for a specified type by providing a non-templated function for that type. For example

```
   double min(double a, double b);
```

This declaration enables us to define a different function for double variables. Like other non-templated functions, standard type conversions (such as promoting a variable of type char to int or variable of type float to double) are supplied.

Note

We use a template when the type T doesn't affect the behavior of the class. If type T does affect the behavior, you will need virtual functions, and you will therefore use inheritance.

9.3 Templates and Macros

In many ways, templates work like preprocessor macros, replacing the templated variable with the given type. However, there are many differences between a macro and a template. For example, consider a macro square that finds the square of an entity.

```
# define square(a)  a * a
```

The corresponding template can be written as

```
template <class T> square (T a)
{
   return (a * a)
}
```

The problems with the square macro is as follows:

1. There is no way for the compiler to verify that the macro parameter is for compatible types. The macro is expanded without any special type checking.
2. A common problem with macros is the errors pointed out by the compiler, if there is any problem with the macro. Since the preprocessor expands macros, compiler error messages will refer to the expanded macro, rather than the macro definition.
3. You will get strange results for the following program:

9.2 Code Listing

```
# include <iostream.h>

# define square(a)  a * a
```

```
void main()
{
   int i, j, k, m = 4;

   i = 16 / square(2);
   j = square(4 + 3);
   k = square(++m);

   cout << i << " " << j << " " << k << " " << m;
}
```

On preprocessing the assignments would be as follows:

```
   i = 16 / 2 * 2;
   j = 4 + 3 * 4 + 3;
   k = ++m * ++m;
```

and the output of the program is **16, 19, 36, 6**

9.4 Class Templates

A class template can be used to create a family of classes that operate on a type. For example

```
template <class T, int i> class tempclass
{
   private :
      T tarray[i];
      int arraysize;

   public :
      tempclass(void);
      ~tempclass(void) ;
      int setdata(T a, int b);
};
```

The template class uses two parameters, a type T and an int i. The T parameter can be passed as any type, including structures and classes. The i parameter has to be passed as an integer constant. Since i is a constant defined at compile time, you can define a member array of size i using a standard automatic array declaration. Unlike function templates, you do not use all template parameters in the definition of a templated class. Members of template class are defined slightly differently than those of non-templated class.

```
template <class T, int i> int tempclass <T, i> :: setdata(T a, int b)
{
   if ((b >= 0) && (b < i))
   {
      tarray[b++] = a;
      return sizeof(a);
   }
   else return -1;
}
```

When instantiating a class template, we must explicitly instantiate the class by giving the parameters for the template class. For example, to create an instance of tempclass we must say:

```
tempclass <float, 6>    tmpobj;          // Fine
tempclass <char, j++)   tmperror;        // Error, second parameter must be a constant
```

Code is not generated for a templated class (or function) unless it's instantiated. Moreover, member functions are instantiated only if they are called. This can cause problems if you are building a library with templates for other users.

One common mistake that occurs while using templates is improper placement of angle brackets. You should use proper spacing and parenthesis to distinguish angle brackets from operators such as >> and **->**. For example:

testclass < int, x > y ? x : y > test;

should be rewritten as:

testclass < int, (x > y ? x : y) > test;

9.3 Code Listing (Template class - Vector)

```
# include <iostream.h>
# include <stdlib.h>

template <class T> class vector
{
   protected:
      int sz;
      T init_value;
      T *ip;

      virtual int getoffset(int x)
      {
         return x;
      }

   public:
      vector(vector<T>& a);
      vector(int size, T def_val);
      vector();
      ~vector();

      int size()
      {
         return sz;
      }

      void resize(int new_sz);
      vector<T> & operator = (vector<T> &);
      virtual T & operator[](int x);
      friend vector<T>& operator + (vector<T> &, vector <T> &);
      friend ostream & operator << (ostream &, vector<T> &);
};

template<class T> vector<T> :: vector()
{
   sz = 0;
   ip = 0;
   init_value = 0;
}

template<class T> vector<T> :: vector(vector<T>& a)
{
   sz = a.sz;
   ip = new T[sz];

   for(int i = 0; i < sz; i++)
      ip[i] = a.ip[i];

   init_value = a.init_value;
```

```
}

template <class T> ostream & operator << (ostream &os, vector<T> &b)
{
   os << "[";

   for(int i = 0; i < b.size(); i++)
   {
      if(i > 0)   os << ",";
      os << b.ip[i];
   }
   return os << "]";
}

template <class T> vector<T> :: vector(int size, T initial)
{
   sz = size;
   init_value = initial;
   ip = new T[size];

   for(int i = 0; i < size; i++)
      ip[i] = initial;
}

template <class T> vector <T> :: ~vector()
{
   delete ip;
}

template <class T> T& vector<T> :: operator[] (int x)
{
   return ip[getoffset(x)];
}

template <class T> vector<T>& vector<T> :: operator = (vector<T> &v)
{
   if(this != &v)
   {
      delete ip;
      ip = new T[sz = v.size()];

      for(int i = 0; i < sz; i++)
         ip[i] = v.ip[i];
   }
   return *this;
}

template <class T> void vector<T> :: resize(int new_size)
{
   T *new_p = new T[new_size];

   for(int i = 0; i < new_size; i++)
      new_p[i] = (i < sz) ? ip[i] : init_value;

   delete ip;
   sz = new_size;
   ip = new_p;
}

template <class T> vector<T>& operator + (vector<T> &a, vector<T> &b)
{
   if(a.size() != b.size())
   {
      cout << "Attempt to add int_vectors of unequal sizes" << endl;
      exit(1);
   }
   vector<T> *c = new vector<T>(a.size(), 0);
   for(int i = 0; i < a.size(); i++)
      c->ip[i] = a.ip[i] + b.ip[i];

   return *c;
}

void main()
{
   vector <int> i(10, 4);
   vector <double> d(5, 4.55);

   cout << i << endl;
   cout << d << endl;

}
```

Note

Implement protected_vector, range_vector and combined_vector as template class. Write a main program to test the same.

9.5 Templates and Static Members

A template can declare static data members. Each instantiation of the template has its own set of static data, one per class type. That is, if you add a static member "count" to the templated vector class to keep a count of how many vector objects have been created, one such member will appear for each type.

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack. Unlike an array, the definition of the stack provides for the insertion and deletion of items, so that a stack is a dynamic, constantly changing object. How does the stack change? A single end of the stack is designated as the stack top. New items may be put on top of the stack (in which case the top of the stack moves upward to correspond to the new highest element), or items that are at the top of the stack may be removed (in which case the top of the stack moves downward to correspond to the new highest element). A stack is a data structure that is LIFO (last in first out), unlike a queue that is FIFO (first in first out).

9.4 Code Listing - Template class (stack) with static members

```
# include <iostream.h>
# include <stdlib.h>

template <class T> class stack
{
   struct node
   {
      T     data;
      node  *next;
   }*p;

   static int countOfObjects;
   static int countOfTypes;

public :
   stack();
   ~stack();
   void push(T);
   T pop();
   void display();
   static int getCountOfTypes()
   {
      return countOfTypes;
   }

   static int getCountOfObjects()
   {
      return countOfObjects;
   }
};

template <class T> int stack<T> :: countOfObjects = 0;
template <class T> int stack<T> :: countOfTypes = 0;

template <class T> stack<T> :: stack()
{
   countOfObjects++;
   p = 0;
}

template <class T> stack<T> :: ~stack()
{
   node* run;
   countOfObjects--;
   while(p != 0)
   {
      run = p;
      p = p->next;
      delete run;
      countOfTypes--;
   }
}

template <class T> void stack<T> :: push(T val)
{
   node*temp = new node;
   temp->data = val;
   temp->next = p;
   p = temp;
   countOfTypes++;
}

template <class T> T stack<T> :: pop()
{
   if(p == 0)  throw "Stack Underflow\n";
   T ret;
   node *temp;
   temp = p;
   p = p->next;
   ret = temp->data;
   delete temp;
   countOfTypes--;
   return ret;
}

template <class T> void stack<T> :: display()
{
   node *run = p;
   while(run != 0)
   {
      cout << run->data << "\t";
      run = run->next;
   }
   cout << endl;
}
void main()
{
   stack <double> ds;
   ds.push(3.33);
   ds.push(1.11);
   ds.push(2.22);
   ds.display();

   cout << "Count of double types = " << stack <double> :: getCountOfTypes();
   cout << "\nCount of stack objects - first = " <<
      stack <double> :: getCountOfObjects();

   stack <int> is;
   is.push(100);
   is.push(200);

   cout << "\nCount of int types = " << stack <int> :: getCountOfTypes();
   cout << "\nCount of stack objects - second " <<
      stack <int> :: getCountOfObjects();

   stack <int> is2;
   is2.push(300);
   is2.push(400);

   cout << "\nCount of int types = " << is2.getCountOfTypes();
   cout << "\nCount of stack objects - third " << is2.getCountOfObjects();

   cout << "\nData in double stack \n";

   try {
      for(int i = 1; ; i++)
         cout << "pop " << i <<" = " << ds.pop() << endl;
   }
   catch(const char* s)
   {
      cout << s << endl;
   }
}
```

9.5 Code Listing - Template class (Single Link List) with static members

/* A single linked list consists of two elements, one is the data element (it is also called node) and the other is the next address. The node contains the actual value, and the next address contains the address of the next element in the list. Such an address that is used to access a particular node is called a pointer. The next address field of the last node in the list contains a special value, known as null, which is not a valid address. This null pointer is used to signal the end of a list. The list with no elements on it is called the empty list or the null list.

A doubly-linked list node contains two links: one pointing to the next node and one to the previous node. They are efficient when we need to process data in both forward and backward direction, however they require an extra link */

```
# include <iostream.h>
# include <stdlib.h>


template <class T> class List
{
   struct node
   {
      T data;
      node* next;
   }*p;

   public:
      List();
      ~List();
      void addatbeg(T);
      void append(T);
      void addbefore(T,T);
      void insertat(int,T);
      void addafter(T,T);
      void del_node(T);
      void display();
};

template <class T> List<T> :: List()
{
   p = 0;
}

template <class T> List<T> :: ~List()
{
   node*run = p;
   while(p != 0)
   {
      run = p;
      p = p->next;
      delete run;
   }
}

template <class T> void List<T> :: display()
{
   node *run = p;
   while(run)
   {
      cout << run->data <<" ";
      run = run->next;
   }
}

template <class T> void List<T> :: addatbeg(T val)
{
   node *temp = new node;
   temp->data = val;
   temp->next = p;
   p = temp;
}

template <class T> void List<T> :: addbefore(T before, T val)
{
   if(p->data == before)
   {
      addatbeg(val);
      return;
   }
   node *old = p;
   node *run = p->next;
   while(run)
   {
      if(before == run->data)
      {
         node *temp = new node;
         temp->data = val;
         temp->next = run;
         old->next = temp;
         return;
      }

      old = run;
      run = run->next;
   }
   cout<<"Required node not found\n";
```

```
}

template <class T> void List<T> :: addafter(T after, T val)
{
   node *run = p;
   while(run)
   {
      if(after == run->data)
      {
         node *temp = new node;
         temp->data = val;
         temp->next = run->next;
         run->next = temp;
         return;
      }
      run = run->next;
   }
   cout<<"Required node not found\n";
}

template <class T> void List<T> :: append(T val)
{
   if(p == 0)
      addatbeg(val);
   else
   {
      node *run = p;
      while(run->next)
         run = run->next;

      run->next = new node;
      run->next->data = val;
      run->next->next = 0;
   }
}

template <class T> void List<T> :: del_node(T del)
{
   node *old = p;
   if(p->data == del)
   {
      p = p->next;
      delete old;
      return;
   }

   node *run = p->next;
   while(run)
   {
      if(run->data == del)
      {
         old->next = run->next;
         delete run;
         return;
      }
      old = run;
      run = run->next;
   }
   cout<<"Required node not found\n";
}

template <class T> void List<T> :: insertat(int at, T val)
{
   if(at == 1)
   {
      addatbeg(val);
      return;
   }
   node *old = p;
   node *run = p->next;
   int i = 2;

   while(run != 0)
   {
      if(i == at)
         break;
      old = run;
      run = run->next;
      i++;
   }

   if(i != at) {
      append(val);
      return;
   }
   node *temp = new node;
   temp->data = val;
   temp->next = run;
   old->next = temp;
}

void main()
{
   List <int> ob;
   ob.addatbeg(10);
   cout<<"\n values in the list are : ";
   ob.display();

   ob.addbefore(10,5);
   cout<<"\n values in the list are : ";
   ob.display();

   ob.addafter(5,2);
   cout<<"\n values in the list are : ";
   ob.display();

   ob.append(20);
   cout<<"\n values in the list are : ";
   ob.display();

   ob.insertat(4,15);
   cout<<"\n values in the list are : ";
   ob.display();

   ob.del_node(5);
   cout<<"\n values in the list are : ";
   ob.display();
}
```

9.6 Exercise

Theory Questions

1. What are parameterized types? How are they useful?

2. What is the difference between a function template and a template function?

3. What is the difference between a class template and a template class?

4. Why do we use templates instead of macros?

5. How do we decide whether to use templates or inheritance?

Problems

1. Implement a template function to swap two variables.

2. Develop a C++ program using function template to perform matrix addition, subtraction and multiplication of two matrices. It should support integer, float and double data types.

3. Implement a template function to find the minimum of three numbers.

4. Implement a template function to find the square and cube of a number.

5. Implement queue using templates.

6. Implement doubly linked list using templates.