C++ Class Templates

Extensible Data Type: a data type that can be modified or extended to be able to handle different or various types of data.

- Class inheritance
  Derive a new class based on an existing class.
  Add new methods or data members.
  Override and redefine existing methods.

- Class Template
  A C++ feature for implementing a class that can be extended to operate on an arbitrarily-specified data type.

  3 C++ Template-oriented mechanisms for dealing with extensible data types:
  - Container: an object that can hold other objects
  - Container Algorithm: acts on containers (e.g., a sorting algorithm)
  - Iterator: a mechanism to sequence through (e.g., traverse) the contents of a container

---

```
template   <class  T>          // template <class T> class MyClass
class  MyClass
{
public:
     MyClass( );                      // constructors
     MyClass( T  initialData) { theData = initialData; };

     // access methods
     void  setData( T  newData) { theData = newData; };
     T  getData( )  { return(theData); };

private:
     T  theData;
};

// ----------------------------------------
int main(int argc, char *argv[])
{
     MyClass<int>         a;
     MyClass<double>      b(5.4);

     a.setData(5);
     cout << b.getData() << endl;
     return(0);
}
```

A "Stack of chars" Class

(with dynamic allocation)

```
#include <assert.h>
const int MaxStack   = 100;           // Used only for default object

class Stack {
public:
   Stack()             {init(MaxStack);}    // default  constructor
   Stack(int request)  {init(request);}     // explicit constructor
   ~Stack();                                // default  destructor
   void push(char  val);                    // add an item
   char pop();                              // remove an item
   bool empty() const  {return(top == -1); }
   bool full()  const  {return(top+1 == stacksize); }

private:
   char *items;                             // Dynamically allocate
   int  stacksize;
   int  top;
   void init(int size);                     // private init routine
};

// ------------------------------------------------------------------------
void Stack::init(int size)                  // initialize a stack object
{
   top = -1;
   stacksize = size;                        // Keep track of declared size
   items     = new char[stacksize];         // allocate space
   assert(items);
}

Stack::~Stack()                             // destructor needed to
{                                           //    release allocated mem.
   delete[] items;
}

void Stack::push(char val)
{
   if( !full() )
      items[ ++top ] = val;
}
char Stack::pop()
{
   if( empty() )  return('\0');
   return( items[top--] );
}

// ------------------------------------------------------------------------
Stack  s1(30);        // Explicitly specify desired stack size
Stack  s2;            // OK to let size default to MaxSize

   s1.push('a');      // Always a stack of chars
```

A Stack Class TEMPLATE

(parameterized with datatype and stacksize)

mystack.h

```
template< class Typ, int MaxStack >
class Stack {
public:
   Stack()           {top = -1;}           // default  constructor
                                           // destructor unnecessary
   void push( Typ  val);                   // add an item
   Typ  pop();                             // remove an item
   bool empty() const;                     // no elements?
   bool full() const;                      // too many elements?

private:
   Typ  items[MaxStack];    // Notice: dynamic allocation not req'd
   int  top;
};
```

---

mystack.cpp

```
template< class Typ, int MaxStack >
void Stack< Typ, MaxStack >::push(Typ  val)
{
   if( !full() )
      items[ ++top ] = val;
}

template< class Typ, int MaxStack >
Typ  Stack< Typ, MaxStack >::pop()
{
   if( empty() )  return('\0');
   return( items[top--] );
}

template< class Typ, int MaxStack >
bool Stack< Typ, MaxStack >::full()
{
   return(top+1 == MaxStack);
}

template< class Typ, int MaxStack >
bool Stack< Typ, MaxStack >::empty()
{
   return(top == -1);
}
```

```
#include "mystack.h"
int main(int argc, char *argv[])
{
Stack<char, 20>   s1;
Stack<int,  100>  s2;

   s1.push('a');                // insert a   on the char  stack
   s2.push(350);                // insert 350 on the int   stack
```

A Stack Class TEMPLATE
Inline implementation
(parameterized with datatype and stacksize)

myStack.h

```
template< class Typ, int MaxStack >
class Stack {
public:
   Stack()                          // default  constructor
      {top = -1;}
   bool  empty() const
      { return(top == -1); }
   bool  full()  const
      { return(top+1 == MaxStack);  }

   void push( Typ  val)             // add an item
   {
      if( !full() )                 // ignore if full stack
         items[ ++top ] = val;
   }

   Typ  pop()                       // remove an item
   {
      if( empty() )
         return(0);                 // .. ??? what to return. Must be "Typ"
      return( items[top--] );
   }

private:
   Typ  items[MaxStack];            // MaxStack size specified via template
   int  top;
};
```

main.cpp

```
#include "mystack.h"

int main(int argc, char *argv[])
{
Stack<char, 20>   s1;
Stack<int,  100>  s2;

   s1.push('a');                    // insert a   on the char  stack
   s2.push(350);                    // insert 350 on the int   stack
```

STL Standard Template Library

Misc:

```
<cassert>       Equivalent to older C-style .h functionality
<cmath>
<cstdlib>            Included names are NOT part of std:: namespace
<cstring>
<ctime>
```

I/O:

```
<iostream>
<istream>
<ostream>            Included names ARE part of std:: namespace
<fstream>
<strstream>
```

Collections:

```
<string>        =, size, compare ops, + concatenate, <<, >>,
                     getline(), c_str()

<vector>        =, size, empty, insert, erase, at, [ ]

<list>          =, empty, size, front, back, push_front, pop_front,
                     push_back, pop_back, insert, erase, remove,
                     sort, merge, swap, reverse, iterate

<stack>         =, size, empty, push, pop, top&

<queue>         =, size, empty, push, pop, front&, back&

<deque>         =, size, empty,
                   push_front, pop_front, push_back, pop_back,
                   insert, erase
```

Exceptions:

```
<exception>     Exception Handlers
<stdexcept>
```

Iterators

Iterator: a mechanism to sequence through (e.g., traverse) the contents of a container

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

int main()
{
   list<int>  myList;
   list<int>::iterator  curr;

   // start the iterator at the beginning of myList
   curr = myList.begin();

   // test for empty list
   if (curr == myList.end())
      cout << "The list is empty" << endl;

   // insert five items into the list myList
   for (int j = 0; j < 5; j++)
   {
      // places item j at the front of the list
      curr = myList.insert(curr, j);   // insert before item curr
                                       // Return iterator to new item
      // myList.push_front(j);
      // myList.push_back(j);
   }

   // now output each item in the list starting with the
   // first item in the list; keep moving the iterator
   // to the next item in the list until the end of
   // the list is reached
   cout << "The List: ";
   for (curr = myList.begin(); curr != myList.end(); curr++)
   {
        cout << *curr << " ";
   }
   cout << endl;

   list<int>::reverse_iterator  revcurr;

   cout << "The List in reverse order: ";
   for (revcurr = myList.rbegin(); revcurr != myList.rend(); revcurr++)
   {
        cout << *revcurr << " ";
   }

   cout << endl;

   return 0;
}
```

```
// Output:
// The list is empty
// The List: 4 3 2 1 0
// The List in Reverse Order: 0 1 2 3 4
```

const_iterator and const_reverse_iterator

Const iterators are allowed to traverse through a list, but are not allowed to be used to modify the contents of a list. A const_iterator must be used to traverse a list that is declared as a const (for example, as a const parameter to a function).

```
////////////////////////////////////////////////////////////////
// Grocery List example: List of strings, iterator, const_iterator
#include <iostream>
#include <list>
#include <string>
using namespace std;

void printList(const list<string> theList);  // prototype

int main(int argc, char *argv[])
{
   list<string>  groceryList;           // create an empty list
   list<string>::iterator  i = groceryList.begin();

   i = groceryList.insert(i, "apples");
   i++;
   i = groceryList.insert(i, "bread");
   i++;
   i = groceryList.insert(i, "juice");

   // groceryList.insert(i, "apples");
   // groceryList.insert(i, "bread");
   // groceryList.insert(i, "juice");

   cout << "Number of items on my grocery list: "
        << groceryList.size() << endl;

   printList(groceryList);
   return 0;
}

void printList(const list<string> theList)
{
   cout << "Items are:" << endl;

   list<string>::const_iterator  listIt;
   listIt = theList.begin();

   while (listIt != theList.end())
   {
      cout << *listIt << endl;
      listIt++;
   }
}
```

```
// Output:
Items are:
apples
bread
juice
```