CSCI 3110
Templated function, Templated class

Templates: C++ mechanism allows a type to be a parameter in the definition of a class or a function

Example 1:

```
template <typename T>      // or template <class T>
void swap(T& x, T& y);

int main()
{
   char s='*', t='$';
   cout << "Before swap, s=" << s << ", t=" << t << endl;
   swap(s, t);
   cout << "After swap, s=" << s << ", t=" << t << endl;

   float u=3.3, v=4.4;
   cout << "Before swap, u=" << u << ", v=" << v << endl;
   swap(u, v);
   cout << "After swap, u=" << u << ", v=" << v << endl;
}

template <typename T>      // or template <class T>
void swap(T& x, T& y)
{
   T temp;

   temp = x;
   x = y;
   y = temp;
}
```

Example 2:

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

template<typename T> T FindSum(T, T);
template<typename T> T FindDiff(T, T);
template<typename T> void PrintResult(T (*Find)(T, T), T, T);

int main()
{
  int n1, n2;
  cout << "enter two integers:" << endl;
  cin >> n1 >> n2;

  PrintResult(FindSum, n1, n2);
  PrintResult(FindDiff, n1, n2);

  float f1, f2;
  cout << "Enter two floats:" << endl;
  cin >> f1 >> f2;

  PrintResult(FindSum, f1, f2);
  PrintResult(FindDiff, f1, f2);
}

template<typename T>
T  FindSum(T n1, T n2)
{
  return (n1+n2);
}

template<typename T>
T FindDiff(T n1, T n2)
{
  return (n1-n2);
}

template<typename T>
void PrintResult(T (*Find)(T, T), T n1, T n2)
{
  cout << "result=" << Find(n1, n2) << endl;
}


/*  This is not allowed:
template < typename T>
typedef T (functionType*)(T, T);

...
template < typename T>
void PrintResult(functionType Find, T n1, T n2) { …}
*/
```

Example 3:

```
template <typename T>
struct listNode
{
   T              data;
   listNode<T>*   next;
   listNode();  //constructor
};  // end struct

template <typename T>
listNode<T>::listNode(): next(nullptr)
{
}  // end default constructor


int main()
{
     listNode<int> iNode;
     listNode<char> chNode ;
    ...
}
```

Non-class type parameter?

```
template <class T, int i>
class buffer
{
        T vec[i];
        int size;
public:
        buffer() : size(i){}
        …
};

buffer<char, 80> charB;
buffer<int, 20>   intB;
```

Example 4

```
template <class T>
class NewClass
{
public:
   NewClass();
   NewClass(T initialData);

   void setData(T newData);
   T getData();
private:
   T theData;
}; // end class


template <class T>
NewClass<T>::NewClass()
{
} // end default constructor

template <class T>
NewClass<T>::NewClass(T initialData) : theData(initialData)
{
} // end constructor

template <class T>
void NewClass<T>::setData(T newData)
{
   theData = newData;
} // end setData

template <class T>
T NewClass<T>::getData()
{
   return theData;
} // end getData

template <class T>
void NewClass<T>::display()
{
   cout << theData;
} // end display

int main()  {
   NewClass<int> first;
   NewClass<double> second(4.8);
    NewClass<int>* p=&first;

   first.setData(5);
   cout << second.getData() << endl;

    NewClass<char> *q=new NewClass<char>;
    q→setData('A');
}
```

Summarize : rules for template class

1. Precede the class definition by the template parameter list.

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

2. Use the generic type names in the template definition to declare data items and member functions.

```
template <typename T>
class stackclass
{ public:
      stackclass();
      ~stackclass();
      T Pop();
      void Push (const T& Item);
    ...............

    Private:
       Node<T> * top;
};
```

3. The template parameter list should precede function definitions.
   Example:

```
template <class T>
void stackclass<T>:: Push (const T& Item)
{  Node<T>* temp = new Node<T>;
   temp→data = item;
   temp→next = top;
   top = temp;
}
```

4. Any reference to the class as a datatype must include the template types enclosed in angle brackets.

```
template <class T>    // assuming the stack is not empty
T stackclass<T>:: Pop ()
{  T temp = top→data;
   return temp;
}
```

5. When declaring instances of a templated class, indicate the actual type to be used for the templated class using angle brackets (<>).

```
int main()          {
  stackclass <int> MyStack;
   .............
  return 0;
}
```

6. A class template can have more than one data-type parameter.

```
template <class T1, class T2>
class studentclass          {
   ...........
};
```

Example 5:

```
const int MAX_STACK = 50;

template <class T>
class stackClass
{
public:
   // constructors and destructor:
   stackClass();  // default constructor
   // copy constructor and destructor are supplied by the compiler

 // stack operations:
   bool StackIsEmpty() const;
   // Determines whether a stack is empty.
   // Precondition: None.
   // Postcondition: Returns true if the stack is empty; otherwise returns false.

   void Push(T NewItem, bool& Success);
   // Adds an item to the top of a stack.
   // Precondition: NewItem is the item to be added.
   // Postcondition: If insertion was successful, NewItem
   // is on the top of the stack and Success is true;  otherwise Success is false.

   void Pop(bool& Success);
   // Removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, the item that was added most recently is removed and
   // Success is true. However, if the stack was empty, deletion is impossible and Success is false.

   void Pop(T & StackTop, bool& Success);
   // Retrieves and removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, StackTop contains the item that was added most recently, the
   // item is removed, and Success is true. However, if the stack was empty, deletion is impossible,
   // StackTop is unchanged, and Success is false.

   void GetStackTop(T & StackTop, bool& Success) const;
   // Retrieves the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, StackTop contains the item that was added most recently and
   // Success is true. However, if the stack was empty, the operation fails, StackTop is unchanged, and
   // Success is false. The stack is unchanged.

private:
   T        Items[MAX_STACK];  // array of stack items
   int      Top;               // index to top of stack
};  // end class


template<class T>
stackClass<T>::stackClass(): Top(-1)
{
}  // end default constructor

template<class T>
```

```
bool stackClass<T>::StackIsEmpty() const
{
  return bool(Top < 0);
}  // end StackIsEmpty

template<class T>
void stackClass<T>::Push(T NewItem, bool& Success)
{
  Success = bool(Top < MAX_STACK - 1);

  if (Success)  // if stack has room for another item
  {  ++Top;
    Items[Top] = NewItem;
  }  // end if
}  // end Push


template<class T>
void stackClass<T>::Pop(bool& Success)
{
  Success = bool(!StackIsEmpty());

  if (Success)  // if stack is not empty,
    --Top;     // pop top
}  // end Pop


template<class T>
void stackClass<T>::Pop(T& StackTop, bool& Success)
{
  Success = bool(!StackIsEmpty());

  if (Success)  // if stack is not empty,
  {  StackTop = Items[Top];  // retrieve top
    --Top;                  // pop top
  }  // end if
}  // end Pop


template<class T>
void stackClass<T>::GetStackTop(T& StackTop,
                          bool& Success) const
{
  Success = bool(!StackIsEmpty());

  if (Success)              // if stack is not empty,
    StackTop = Items[Top];  // retrieve top
}  // end GetStackTop
// End of implementation file.
```

```
#include "stack.cpp"
#include <iostream>
Using namespace std;
int main()
{
  bool Success;
```

```
  stackClass<char> charStack;

  charStack.Push('h', Success);
  charStack.Push('a', Success);
  charStack.Push('p', Success);
  charStack.Push('p', Success);
  charStack.Push('y', Success);

  char chValue;
  while (!S_char.StackIsEmpty())
  {
     charStack.Pop(chValue, Success);
     cout << chValue ;
  }
  cout << endl;

  stackClass<int> intStack;

  intStack.Push(2, Success);
  intStack.Push(4, Success);
  intStack.Push(6, Success);
  intStack.Push(8, Success);
  intStack.Push(10, Success);

  int iValue;
  while (!intStack.StackIsEmpty())
  {
     intStack.Pop(iValue, Success);
     cout << iValue << "  ";
  }
  cout << endl;
}
```

Example 6

```
template <class T> struct listNode;  // defined in ListT.cpp

template <class T>
class listClass
{
public:
// constructors and destructor:
   listClass();
   listClass(const listClass<T>& L);
   virtual ~listClass();

// list operations:
   virtual bool ListIsEmpty() const;
   virtual int ListLength() const;
   virtual void ListInsert(int NewPosition,  NewItem,  bool& Success);
   virtual void ListDelete(int Position, bool& Success);
   virtual void ListRetrieve(int Position, & DataItem, bool& Success) const;
   virtual void DisplayList() const;

protected:
   void SetSize(int NewSize);
   listNode<T>* ListHead() const;
   void SetHead(listNode<T>* NewHead);
   T ListItem(listNode<T>* P) const;
   listNode<T>* ListNext(listNode<T>* P) const;

private:
   int          Size;
   listNode<T>* Head;

   listNode<T>* PtrTo(int Position) const;
};  // end class
// End of header file.

template <class T>
struct listNode
{
   T           Item;
   listNode<T>* Next;
   listNode();  //constructor
};  // end struct

template <class T>
listNode<T>::listNode(): Next(NULL)
{
}  // end default constructor

template <class T>
listClass<T>::listClass(): Size(0), Head(NULL)
{
}  // end default constructor


template <class T>
void listClass<T>::ListInsert(int NewPosition, T NewItem,  bool& Success)
```

```
{
  int NewLength = ListLength() + 1;

  Success = bool( (NewPosition >= 1) &&
              (NewPosition <= NewLength) );

  if (Success)
  {  Size = NewLength;

 // create new node and place NewItem in it
    listNode<T>* NewPtr = new listNode<T>;
    Success = bool(NewPtr != NULL);

    if (Success)
    {  NewPtr->Item = NewItem;

      // attach new node to list
      if (NewPosition == 1)
      {  // insert new node at beginning of list
        NewPtr->Next = Head;
        Head = NewPtr;
      }

      else
      {  listNode<T>* Prev = PtrTo(NewPosition-1);
        // insert new node after node
        // to which Prev points
        NewPtr->Next = Prev->Next;
        Prev->Next = NewPtr;
      }  // end if
    }  // end if
  }  // end if
} // end ListInsert

template <class T>
listNode<T>* listClass<T>::PtrTo(int Position)
{
  if ( (Position < 1) || (Position > ListLength()) )
   return NULL;

  else  // count from the beginning of the list
  {  listNode<T>* Trav = Head;
    for (int Skip = 1; Skip < Position; ++Skip)
      Trav = Trav->Next;
    return Trav;
  }
}  // end PtrTo


#include "list.h"
int main()
{
  listClass<double> FloatList;
  listClass<char>   CharList;
  bool              Success;
```

```
   FloatList.ListInsert(1, 1.1, Success);
   FloatList.ListInsert(2, 2.2, Success);

   CharList.ListInsert(1, 'a', Success);
   CharList.ListInsert(2, 'b', Success);
}
```

derive a templated sorted list based on the templated list class

```
#include "listclass.cpp"

template<class T>
class sortedList : public listClass<T>
{
   public:
            sortedList();
            virtual void SortedListInsert(T NewItem, bool& Success); // or simply named "ListInsert"
           ~sortedList();

};

template<class T>
sortedList<T>::sortedList():listClass<T>(){}

template<class T>
void sortedList<T>::SortedListInsert(T newItem, bool&success)
{
    T dataItem;
    bool done=false;

    int i;
    for (i=1; i<=(*this).ListLength(); i++)
    {
       (*this).ListRetrieve(i, dataItem, success);

             if (newItem < dataItem)
                   break;
    }
    (*this).ListInsert(i, newItem, success);  // If using the same name as "ListInsert" in derived class:
                                              // call base class list insert function using:
                                              // (*this).listclass::ListInsert(i, newItem, success);
                                              // or this->listclass::ListInsert(i, newItem, success);
}

template<class T>
sortedList<T>::~sortedList(){}

int main()
{
  sortedList<int>  list1;
  bool success;

  list1.SortedListInsert(4, success);
  list1.SortedListInsert(2, success);
  list1.SortedListInsert(10, success);
  list1.SortedListInsert(0, success);
  list1.DisplayList();

  sortedList<char> list2;
  list2.SortedListInsert('h', success);
  list2.SortedListInsert('k', success);
  list2.SortedListInsert('a', success);
  list2.SortedListInsert('j', success);
```

```
  list2.DisplayList();
  return 0;
}
```