Linked List

Templates

1 Templates

Templates allow us to create *generic* code, i.e., one code base for many/all data types, instead of code for each particular data type.

A template can be defined for:

- Function
- Class

1.1 Template Function

Consider the case of squaring a number (a simple function):

```
int square( int x ) {
   return x * x;
}

double square( double x ) {
   return x * x;
}

int main()
{
  cout << square(3) << endl;
  cout << square(3.1) << endl;

  return 0;
}
```

Issue(s):

   Repeated code (functions).

One solution: Use macros. Good/Bad?

```
#define SQUARE(x) (x*x)
#define MIN(X,Y)  ((X) < (Y) ? (X) : (Y))

int main()
{
  cout << SQUARE(3) << endl;
  cout << SQUARE(3.1) << endl;
  cout << SQUARE(3.1e-06) << endl;

  cout << "min: " << MIN(3,MIN(3.1,3.1e-06)) << endl;

  return 0;
}
```

By convention, macro names are defined using ALL capital letters.

Issue(s):
Type checking/safety?

Another solution: Use templates. Good/Bad?

```
template <typename T>
T square( T x ) {
   return x * x;
}

int main()
{
  cout << square<int>(3) << endl;
  cout << square<double>(3.1) << endl;

  return 0;
}
```

Issue(s):

Code bloat in some cases.

1.2 Template Class

String class that handles different languages:

```
template<class C>  class String
{
    struct Srep;
    Srep *rep;
public:
    String();
    String( const C* );
    String( const String& );

    C read( int i ) const;
    // ...
};
```

Some details:

- **All code** is in a header file!
- The prefix `template<class C>` specifies that a template is being declared and that an argument `C` of data type *type* will be used in the declaration.
- After `C` is introduced, it is used exactly like any other type name.
- The scope of `C` extends to the end of the declaration prefixed by `template<class C>`.
- Note that `template<class C>` says that `C` is a *type* name; it need not be the name of a *class*.

The name of a class template followed by a type, bracketed by `<  >`, is the name of a class (as defined by the template) and can be used exactly like other class names.

```
String<char> cs;
String<wchar_t> ws;

class Jchar {
    //  Japanese character
};

String<Jchar> js;
```

1.3 STL — Brief Overview

The Standard Template Library (STL) is a library of standard class and function templates.

The STL contains six kinds of components:

- containers,
- container adapters,
- iterators,
- algorithms,
- functors (function objects), and
- function adapters

Some references break the STL into only four components, the adapters are not considered separate components.
`http://en.wikipedia.org/wiki/Standard_Template_Library`

Most of the material we have covered (or will cover) is part of the STL (i.e., stacks, queues, and trees).

2 Linked List Template Class

```
/*  linkListT.h

    This is a class for a sorted linked list of type
    "LLT".
    The data type LLT must allow a > comparison.
 */

#include <bool.h>
#include <iostream.h>


template <class LLT>
class LinkedList
{
private:

  struct node
  {
      LLT info;
      node* next;
  };

  typedef node* nodeptr;

  nodeptr head;

  int count;
```

```
public:

    LinkedList()        // Constructor
    {
        head = NULL;
        count = 0;
    }

    ~LinkedList()       // Destructor
    {
        nodeptr p = head, n;

        while( p != NULL )
        {
            n = p;
            p = p->next;
            delete n;
        }
    }

    void AddNode( LLT x );

    void DeleteNode( LLT x );

    void PrintNodes();

    bool IsInList( LLT x );

    int Size();
};
```

```
template <class LLT>
void LinkedList<LLT>::AddNode(LLT x)
{
    nodeptr n, prev, curr;

    n = new node;

    n->info = x;
    n->next = NULL;
    count++;

    if( head == NULL )
        head = n;
    else
    {
        curr = head;
        while( curr != NULL && x > curr->info ) {
            prev = curr;
            curr = curr->next;
        }

        if( curr == head ) {
            n->next = head;
            head = n;
        }
        else {
            prev->next = n;
            n->next = curr;
        }
    }
}
```

```
template <class LLT>
void LinkedList<LLT>::DeleteNode(LLT x)
{
    nodeptr prev, curr;

    curr = head;

    while( curr != NULL && x > curr->info )
    {
        prev = curr;
        curr = curr->next;
    }

    if( x == curr->info )
    {
        if ( curr == head )
            head = head->next;
        else
            prev->next = curr->next;

        delete curr;
        count--;
    }
}
```

```
template <class LLT>
void LinkedList<LLT>::PrintNodes()
{
    nodeptr p = head;

    while( p != NULL )
    {
       cout << p->info << endl;
       p = p->next;
    }
}

template <class LLT>
bool LinkedList<LLT>::IsInList(LLT x)
{
    nodeptr p = head;

    while( p != NULL && x > p->info )
        p = p->next;

    return (x == p->info);
}

template <class LLT>
int LinkedList<LLT>::Size()
{
    return count;
}
```

2.1 Test Code

```
/*  testLink.cpp
 */

#include <iostream.h>

#include "linkListT.h"

    //  prototypes
void  TestIntegerList();
void  TestDoubleList();

int main()
{
    cout << "Testing integer list:\n" << endl;
    TestIntegerList();

    cout << "\n------------------------------------\n" << endl;

    cout << "Testing real list:\n" << endl;
    TestDoubleList();

    return 0;
}
```

```
void  TestIntegerList()
{
    LinkedList<int> list1 ;

      //  add some initial nodes
    list1.AddNode( 3 );
    list1.AddNode( 5 );

    cout << "Initial contents of list1:" << endl;
    list1.PrintNodes();

      //  add a few more nodes
    list1.AddNode( 1 );

    cout << "Contents of list1 after adding:" << endl;
    list1.PrintNodes();

      //  delete a few nodes
    list1.DeleteNode( 5 );

    cout << "Contents of list1 after deleting:" << endl;
    list1.PrintNodes();
}
```

```
void  TestDoubleList()
{
    LinkedList<double> listD ;

      //  add some initial nodes
    listD.AddNode( 3.3 );
    listD.AddNode( 5.4 );
    listD.AddNode( 7.2 );

    cout << "Initial contents of listD:" << endl;
    listD.PrintNodes();

      //  add a few more nodes
    listD.AddNode( 1.1 );

    cout << "Contents of listD after adding:" << endl;
    listD.PrintNodes();

      //  delete a few nodes
    listD.DeleteNode( 5.4 );
    listD.DeleteNode( 3.3 );

    cout << "Contents of listD after deleting:" << endl;
    listD.PrintNodes();
}
```

2.2 Sample Output

```
Testing integer list:

Initial contents of list1:
3
5
Contents of list1 after adding:
1
3
5
Contents of list1 after deleting:
1
3

------------------------------------

Testing real list:

Initial contents of listD:
3.3
5.4
7.2
Contents of listD after adding:
1.1
3.3
5.4
7.2
Contents of listD after deleting:
1.1
7.2
```