**C++ Templates**

Templates allow us to create data structures without defining the type of data that lives in the structure.

**List Node Template**

For example, we could make a list data structure. It could be a list of integers, a list of floats, a list of MyClass, and so on. C++ allows us to make the list structure without defining the type of data inside it.

Here's a list node containing data whose type is some *unspecified* class T. This is a template.

template< typename T >

class Lnode {

public:

T data;

private:

Lnode< T > \*next;

};

In the declaration of ``Lnode'' above, there's a public part that is data of type T and a private part that is a pointer to an Lnode of type T.

In declaring a variable from a template class, we must specify the *type* of data, T, in the template class. To create a list node N containing an integer, we use

Lnode<int> N;

To create one containing an instance of MyClass, we use

Lnode<MyClass> N;

A pointer to a list node of type T would be declared as

Lnode<T> \*p;

**Linked List Template**

It's no use having a list node without a linked list. Here's a linked list template:

template<typename T>

class List {

public:

int empty();

void add( T data );

T remove();

List() {

sentinel = new Lnode<T>();

sentinel->next = sentinel;

}

private:

Lnode<T>\* sentinel;

};

This class supports three functions: empty() returns 0 or 1; add(data) adds data of type T to the list; remove() removes data from the list and returns it (the return type is T). The constructor List() creates a sentinel node and points it to itself.

Some things to note:

* ``T'' is used wherever the list data type is needed.
* A function in this class can have data of type T as an argument and can return data of type T.
* When creating a new Lnode, we have to make sure it's of the correct type, so we call ``new Lnode< T >()'', which create an Lnode of type T.

**Function Definitions for Template Classes**

Recall that functions defined *inside* the class declaration are compiled ``in-line'', which is to say their code is put in place of a function call. This takes more space, but saves the time of the function call. For this reason, we typically put *large* functions outside the class defintion.

Here's how we might define the add() function *outside* the template class declaration. This just creates a new Lnode and adds it to the head of the list.

template<typename T>

void List<T>::add(T data)

{

Lnode<T> \*p;

p = new Lnode<T>();

p->data = data;

p->next = sentinel->next;

sentinel->next = p;

}

Template code can get a bit messy, as you can see above. However, the only differences between it and normal code are that

* template<typename T> appears above the class or function declaration, and
* <T> appears after any template class (List or Lnode in this case) that depends upon T.

Here's how we might define the remove() function.

template<typename T>

T List<T>::remove()

{

T data;

Lnode<T> \*node;

if (sentinel->next == sentinel) {

cerr << "ERROR: `remove' called with empty list.\n";

exit(1);

}

node = sentinel->next;

data = node->data;

sentinel->next = node->next;

delete node;

return data;

}

**Using Template Classes in Your Code**

Template classes are used much like normal classes. The *only* difference is that declaration of a class instance must include the template type, T. This is great from the point of view of someone using the template, since they can easily use the template on various data types.

#include "list.h"

main()

{

List<int> list1;

List<float> list2;

list1.add( 5 );

list2.add( 2.7 );

cout << list1.remove();

... etc ...

}