Class and Function Templates

Motivation for Templates

You want both:

- a list of `Location` objects
- a list of `MazeMonster` objects

How can you accomplish this by writing one `LinkedList` class?

- state all the ways you can think of doing this
- state the pros/cons of each method

One Way to Look at Templates...

Until now, we have used variables:

- The type of a variable is fixed when you write code.
- The value of a variable isn't fixed when you write code.

With templates, type isn't fixed when you write code!

With templates, you use a type more or less as a variable!

Example: Queue of some type Foo

C++ keywords

```
template <class Foo> class Queue {
```

Parameter,
can be any type

```
private:
   Foo buffer[100];
   int head, tail, count;

public:
   Queue();
   void Insert(Foo item);
   Foo Remove();
   ~Queue();
};
```

The header of a templated class declaration specifies one or more type names which are then used within the template declaration.

These type names are typically NOT standard or user-defined types, but merely placeholder names that serve as formal parameters.

C++ Templates

Definition of "template":

Parameterized class with parameters that denote unknown types.

Usage:

In situations where the same algorithms and/or data structures need to be applied to different data types.

Declaration syntax:

```
template <class Foo> class Queue {
// template member declarations go here
};
```

What can a parameter be used for?

To specify the type specifying of data which will be local to objects of the class:

```
private:
   Foo buffer[100];
```

To specify the type of a parameter to a class member function:

```
void Insert(Foo item);
```

To specify the return type of a class member function:

```
Foo Remove();
```

Instantiating a Template

Given the template declaration:

```
template <class Foo> class Queue {...};
```

Instantiate Queue of ints in 2 ways:

- `Queue<Location>` **`intQueue`**`;`
- `typedef Queue<int> IntegerQueue;`
  `IntegerQueue` **`intQueue`**`;`

Both of these define an **object** `intQueue`.

Note how an actual type (`Location` or `int`) is substituted for the template parameter (`Foo`) in the object declaration.

Usage of Templates

Once created, the template object is used like any other object:

```
intQueue.Insert(100);     // add 100 to the queue
intQueue.Insert(200);     // add 200
```

The parameter type for the member function `Insert()` was specified as `Foo` in the template declaration and mapped to `int` in the declaration of the object `intQueue`. When calling `Insert()` we supply an `int` value.

```
int x = intQueue.Remove(); // remove 100
intQueue.Insert(300);      // queue now
                           //    has (200,300)
int Sz = intQueue.Size();  // size is 2
```

Compiler view of templates...

The compiler macro expands the template code:

- You write `Queue<int> intQueue.`
- Compiler emits new copy of a class named "Queueint" and substitutes `<int>` for `<Foo>` throughout.
- Therefore, the compiler must have access to the implementation of the template member functions in order to carry out the substitution.
- Therefore, the template implementation CANNOT be pre-compiled.
- Most commonly, all template code goes in the header file with the template declaration.

The compiler "maps" the declaration:

```
private:
      Foo buffer[100];
```

to the declaration:

```
private:
   Queueint buffer[100];
```

The compiler "mangles" the template name with the actual parameter (type name) to produce a unique name for the class.

Implementing Template Class Methods

`template` and actual parameter(s)

Class name and actual parameter(s)

Scope resolution operator and function name

```
template<class Foo> Queue<Foo>::Queue() {
  // ... member function body goes here
}
```

Return type goes here:

```
template<class Foo> void Queue<Foo>::Insert(Foo item) {
  // ... member function body goes here
}
```

A Complete Template Queue Class

```
// QueueT.h
#ifndef QUEUET_H
#define QUEUET_H
#include <cassert>
const int Size = 100;
template <class Foo> class QueueT {
private:
   Foo buffer[Size];
   int  Head, Tail, Count;
public:
   QueueT();
   void Enqueue(Foo Item);
   Foo Dequeue();
   int  getSize() const;
   bool isEmpty() const;
   bool isFull() const;
   ~QueueT();
};
// . . . template implementation goes here
#endif
```

Using the template parameter: data type, parameter type, return type.

A Complete Template Class


```
// . . . continuing header file QueueT.h

template <class Foo> QueueT<Foo>::QueueT() : Head(0),
                                  Tail(0), Count(0) {
}

template <class Foo> void QueueT<Foo>::Enqueue(Foo Item) {

   assert(Count < Size);     // die if Queue is full!

   buffer[Tail] = Item;
   Tail = (Tail + 1) % Size; // circular array indexing
   Count++;
}
```

A Complete Template Class


```
// . . . continuing header file QueueT.h

template <class Foo> Foo QueueT<Foo>::Dequeue() {

   assert(Count > 0);        // die if Queue is empty

   int oldHead = Head;       // remember where old Head was
   Head = (Head + 1) % Size; // reset Head
   Count--;
   return buffer[oldHead];   // return old Head
}

template <class Foo> int QueueT<Foo>::getSize() const {

   return (Count);
}
```

A Complete Template Class

```
// . . . continuing header file QueueT.h

template <class Foo> bool QueueT<Foo>::isEmpty() const {

   return (Count == 0);
}

template <class Foo> bool QueueT<Foo>::isFull() const {

   return (Count < Size - 1);
}

template <class Foo> QueueT<Foo>::~QueueT() {
}

// . . . end template QueueT<Foo> implementation
```

A Driver for the QueueT Template


```
#include <iostream>
#include <iomanip>
using namespace std;
#include "QueueT.h"

void main() {

   const int numVals = 10;
   QueueT<int> intQ;

   for (int i = 0; i < numVals; i++) {
      intQ.Enqueue(i*i);
   }

   int Limit = intQ.getSize();
   for (i = 0; i < Limit; i++) {
      int nextVal = intQ.Dequeue();
      cout << setw(3) << i << setw(5) << nextVal << endl;
   }
}
```

```
  0    0
  1    1
  2    4
  3    9
  4   16
  5   25
  6   36
  7   49
  8   64
  9   81
```

Recap

Note that method bodies use the same algorithms for a queue of `int`s or a queue of `double`s or a queue of `Location`s…

But the compiler still type checks!

It does a macro expansion, so if you declare

```
QueueT<int>       iQueue;
QueueT<char>      cQueue;
QueueT<Location> Vertices;
```

the compiler has three different classes after expansion to use with normal type checking rules.

Implicit Assumptions in QueueT

Declaration of the array of `Foo`s assumes `Foo` has a default constructor:

```
template <class Foo> class Queue {
   private:
      Foo buffer[Size];
      ...
};
```

Assignment of `Foo`s assumes `Foo` has appropriately overloaded the assignment operator:

```
template <class Foo> void Queue<Foo> ::Insert(Foo item) {
   ...
   buffer[tail] = item;
   ...
};
```

Implicit Assumptions in QueueT

The way that `Foo`s are returned by `Remove()` method assumes `Foo` has provided an appropriate copy constructor:

```
template <class Foo> Foo Queue<Foo>::Remove() {
   ...
   return buffer[val];
}
```

Variable and Constant Template Parameters

Template parameters may be:

type names	(we saw this previously)
variables	e.g., to specify a size for a data structure
constants	useful to define templates for special cases (not terribly useful)

Queue Template with a Variable Parameter

One weakness of the QueueT template is that the queue array is of a fixed size. We can easily make that user-selectable:

```
template <class Foo, int Size> class QueueT {
private:
   Foo  buffer[Size];
   int  Head,
        Tail;
   int  Count;
public:
   QueueT();
   bool Enqueue(Foo Item);
   bool Dequeue(Foo& Item);
   int  getSize() const;
   bool isEmpty() const;
   bool isFull() const;
   ~QueueT();
};
```

Second template parameter is just an `int` variable, which falls within the class scope just as a private data member would.

Driver for Revised Queue Template


```
#include <iostream>
#include <iomanip>
using namespace std;
#include "QueueT.h"
void main() {
   const int smallSize =  10;
   const int largeSize = 100;

   QueueT<int, smallSize> smallQ;
   QueueT<int, largeSize> largeQ;

   for (int i = 0; i < smallSize-1; i++)
      smallQ.Enqueue(i);

   for (i = 0; i < largeSize-1; i++) {
      largeQ.Enqueue(i);

   for (i = 0; i < smallSize-1; i++) {
      int nextVal;
      largeQ.Dequeue(nextVal);
      cout << setw(3) << i << setw(5) << nextVal << endl;
   }
}
```

The value specified in the declaration must still be a constant though…

… that could be avoided by redesigning the template to take the array size as a parameter to a constructor…

Template Type-checking

Suppose we have the declarations:

```
QueueT<int,    100>   smallIntegerQueue;
QueueT<int,   1000>   largeIntegerQueue;
QueueT<int,   1000>   largeIntegerQueue2;
QueueT<float,  100>   smallRealQueue;
QueueT<float, 1000>   largeRealQueue;
```

Which (if any) of the following are legal assignments:

```
smallIntegerQueue = largeIntegerQueue;

smallIntegetQueue = smallRealQueue;

largeIntegerQueue = largeIntegerQueue2;
```

Linked List Template


```
// LinkListT.h
#ifndef LINKLISTT_H
#define LINKLISTT_H

#include <cassert>
#include "LinkNodeT.h"

template <class ItemType> class LinkListT {
private:
   LinkNodeT<ItemType>* Head;  // points to head node in list
   LinkNodeT<ItemType>* Tail;  // points to tail node in list
   LinkNodeT<ItemType>* Curr;  // points to "current" node in list

public:
   LinkListT();
   LinkListT(const LinkListT<ItemType>& Source);
   ~LinkListT();

. . .
```

These are pointers to template objects so any template parameters must be specified explicitly.

Function parameter is a template object, so…

Linked List Template


```
. . .
bool isEmpty() const;
   bool moreList() const;
   bool PrefixNode(ItemType newData);
   bool AppendNode(ItemType newData);
   bool InsertAfterCurr(ItemType newData);
   bool Advance();
   void gotoHead();
   void gotoTail();
   bool MakeEmpty();
   bool DeleteCurrentNode();
   bool DeleteValue(ItemType Target);
   ItemType getCurrentData() const;
   void PrintList(ostream& Out);
   LinkListT<ItemType>& operator=(const LinkListT<ItemType>&
                                  Source);
};

#include "LinkListT.cpp"

#endif
```

Operator return type is a template object, so…

Linked List Node Template

```
// LinkNodeT.h
#ifndef LINKNODET_H
#define LINKNODET_H

template <class ItemType> class LinkNodeT {
private:
   ItemType Data;
   LinkNodeT<ItemType>* Next;

public:
   LinkNodeT();
   LinkNodeT(ItemType newData);
   void setData(ItemType newData);
   void setNext(LinkNodeT<ItemType>* newNext);
   ItemType getData() const;
   LinkNodeT<ItemType>* getNext() const;
};

#include "LinkNodeT.cpp"

#endif
```

Function return type is a template object, so…

A Node Template Constructor

```
////////////////////////////////////////////////////
// Constructor for LinkNode objects with assigned
// Data field.
//
// Parameters:
//   newData   Data element to be stored in node
// Pre:        none
// Post:       new LinkNode has been created with
//                 given Data field and NULL
//                 pointer
//
template <class ItemType>
LinkNodeT<ItemType>::LinkNodeT(ItemType newData) {
   Data = newData;
   Next = NULL;
}
```


Node Template Mutators

```
/////////////////////////////////////////////////////
// Sets new value for Data element of object.
//
// Parameters:
//   newData   Data element to be stored in node
// Pre:        none
// Post:       Data field of object has been
//                 modified to hold newData
//
template <class ItemType>
void LinkNodeT<ItemType>::setData(ItemType newData) {

   Data = newData;
}

/////////////////////////////////////////////////////
// Suppressed to save space.
//
template <class ItemType>
void LinkNodeT<ItemType>::setNext(LinkNodeT<ItemType>* newNext) {

   Next = newNext;
}
```

Node Template Reporters

```
////////////////////////////////////////////////////
// Returns value of Data element of object.
//
// Parameters: none
// Pre:        object has been initialized
// Post:       Data field of object has been
//                 returned
//
template <class ItemType>
ItemType LinkNodeT<ItemType>::getData() const {

   return Data;
}

////////////////////////////////////////////////////
// Suppressed to save space.
//
template <class ItemType>
LinkNodeT<ItemType>* LinkNodeT<ItemType>::getNext() const {

   return Next;
}
```

Linked List Template Destructor

```
////////////////////////////////////////////////////
// Destructor for LinkListT objects.
//
// Parameters: none
// Pre:        LinkListT object has been constructed
// Post:       LinkListT object has been destructed;
//                 all dynamically-allocated nodes
//                 have been deallocated.
//
template <class ItemType> LinkListT<ItemType>::~LinkListT() {

   LinkNodeT<ItemType>* toKill = Head;

   while ( toKill != NULL) {
      Head = Head->getNext();
      delete toKill;
      toKill = Head;
   }
}
```

Linked List Template Prefix Function

```
////////////////////////////////////////////////////////////////
// Inserts a new LinkNodeT at the front of the list.
//
template <class ItemType> bool
LinkListT<ItemType>::PrefixNode(ItemType newData) {

   LinkNodeT<ItemType>* newNode =
                                new LinkNodeT<ItemType>(newData);

   if (newNode == NULL) return false;

   if ( isEmpty() ) {
      newNode->setNext(NULL);
      Head = Tail = Curr = newNode;
      return true;
   }
   newNode->setNext(Head);
   Head = newNode;

   return true;
}
```

Linked List Template Assignment Overload

```
//////////////////////////////////////////////////////////////
// Deep copy assignment operator for LinkListT objects.
//
template <class ItemType>
LinkListT<ItemType>& LinkListT<ItemType>::
operator=(const LinkListT<ItemType>& Source) {

   if (this != &Source) {

      MakeEmpty();                // delete target's list, if any

      LinkNodeT<ItemType>* myCurr = Source.Head; // copy list

      while (myCurr != NULL) {
         ItemType xferData = myCurr->getData();
         AppendNode(xferData);
         myCurr = myCurr->getNext();
      }
   }
   return *this;
}
```

Linked List Template Copy Constructor

```
//////////////////////////////////////////////////////////////
// Deep copy constructor for LinkListT objects.
//
template <class ItemType>
LinkListT<ItemType>::LinkListT(const LinkListT<ItemType>& Source) {

   Head = Tail = Curr = NULL;

   LinkNodeT<ItemType>* myCurr = Source.Head;       // copy list

   while (myCurr != NULL) {
      ItemType xferData = myCurr->getData();
      AppendNode(xferData);
      myCurr = myCurr->getNext();
   }
}
```

Template Functions

The template mechanism may also be used with non-member functions:

```
template <class Foo> Swap(Foo& First, Foo& Second) {
   Foo tmpFoo = First;
   First = Second;
   Second = tmpFoo;
}
```

Given the template function above, we may swap the value of two variables of ANY type, provided that a correct assignment operation and copy constructor are available.

However, the two actual parameters MUST be of exactly the same type:

```
double X = 3.14159;
int    a = 5;
Swap(a, X);           // error at compile time
```

Template Sort Function

```
template <class Foo> InsertionSort(Foo* const A, int Size) {

   int Begin, Look;
   Foo Item;

   for (Begin = 1; Begin < Size; Begin++) {
      Look = Begin - 1;
      Item = A[Begin];
      while ( Look >= 0 && A[Look] > Item) {
         A[Look + 1] = A[Look];
         Look--;
      }
      A[Look + 1] = Item;
   }
}
```

This will use the insertion sort algorithm to sort an array holding ANY type of data, provided that there are > and deep = operators for that type (if a deep assignment is logically necessary).