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

```c++
template <class Foo> class QueueT {
  private:
    Foo Buffer[100];
    int Head, Tail, Count;
  public:
    QueueT();
    void Enqueue(const Foo& item);
    Foo Dequeue();
  ~QueueT();
};
```

Name of template

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 formal 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:
```c++
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:

```cpp
private:
    Foo Buffer[100];
```

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

```cpp
void Enqueue(const Foo& item);
```

To specify the return type of a class member function:

```cpp
Foo Dequeue();
```

Instantiating a Template

Given the template declaration:
```
template <class Foo> class QueueT {...};
```

Instantiate QueueT of objects in two ways:

- `QueueT<Location>`
  ```cpp
  LocationQueue
  intQueue;
  ```
- `typedef QueueT<int> IntegerQueue;`
  ```cpp
  IntegerQueue intQueue;
  ```

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

Usage of Templates

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

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

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

```cpp
int x = intQueue.Dequeue(); // remove 100
int y = intQueue.Enqueue(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 `QueueT<int>` `intQueue`.
- Compiler emits new copy of a class named, say, "Queueint" and substitutes "int" for "Foo" throughout.
- Therefore, the compiler must have access to the parameterized 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:

```cpp
private:
    Foo Buffer[100];
```

to the declaration:

```cpp
private:
    int Buffer[100];
```
Templates

### Implementing Template Class Methods

- **Template and formal parameter(s):**
  - `template <class Foo>`

- **Class name and formal parameter(s):**
  - `QueueT<Foo>`

- **Scope resolution operator and function name:**
  - `::Queue()`

- **Return type goes here:**
  - `void Enqueue(Foo item)`

---

### A Complete Template Queue Class

```cpp
// 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(const Foo& item);
  Foo Dequeue();
  int getSize() const;
  bool isEmpty() const;
  bool isFull() const;
  ~QueueT();
};
#endif
```

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

---

```cpp
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++;
}
template <class Foo> void QueueT<Foo>::Enqueue(Foo item) {
  // ... member function body goes here
}
```

---

```cpp
template <class Foo> Foo QueueT<Foo>::Dequeue() {
  assert(Count > 0); // die if Queue is empty
  int oldHead = Head;
  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);
}

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;
    }
}

Recap

Note that method bodies use the same algorithms for a queue of ints or a queue of doubles or a queue of Locations...

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 instantiated three different classes after expansion to use with normal type checking rules.

Implicit Assumptions in QueueT

Declaration of the array of Foos assumes Foo has a default constructor:

    template <class Foo> class QueueT {
        private:
            Foo Buffer[Size];
    };

Assignment of Foos assumes Foo has appropriately overloaded the assignment operator:

    template <class Foo> void QueueT<Foo>::Enqueue(Foo item) {
        ... Buffer[tail] = item;
    }...
Implicit Assumptions in QueueT

The way that Foos are returned by Dequeue() method assumes Foo has provided an appropriate copy constructor:

```cpp
template <class Foo> Foo QueueT<Foo>::Dequeue() {
... return Buffer[oldHead];
}
```

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:

```cpp
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();
};
```

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

```cpp
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();
};
```

Driver for Revised Queue Template

```cpp
#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 < largeSize-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…
Suppose we have the declarations:

```c++
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;`
- `smallIntegerQueue = smallRealQueue;`
- `largeIntegerQueue = largeIntegerQueue2;`

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

Function return type is a pointer to a template object, so...

```
bool DeleteValue(Item Target);
```

Operator return type is a template object, so...

```
operator=(const LinkListT<Item>& Source);}
```

Function parameter is a template object, so...

```
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 Item> LinkNodeT<Item>::LinkNodeT(Item 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 Item> void LinkNodeT<Item>::setData(Item newData) {
    Data = newData;
}

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 Item> Item LinkNodeT<Item>::getData() const {
    return Data;
}

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 Item> LinkListT<Item>::~LinkListT() {
    LinkNodeT<Item>* toKill = Head;
    while (toKill != NULL) {
        Head = Head->getNext();
        delete toKill;
        toKill = Head;
    }
}
**Linked List Template Prefix Function**

```cpp
// Inserts a new LinkNodeT at the front of the list.

template <class Item> bool LinkListT<Item>::PrefixNode(Item newData) {
    LinkNodeT<Item>* newNode = new LinkNodeT<Item>(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**

```cpp
// Deep copy assignment operator for LinkListT objects.

template <class Item> LinkListT<Item>& LinkListT<Item>::operator=(const LinkListT<Item>& Source) {
    if (this != &Source) {
        MakeEmpty(); // delete target's list, if any
        LinkNodeT<Item>* myCurr = Source.Head; // copy list
        while (myCurr != NULL) {
            Item xferData = myCurr->getData();
            AppendNode(xferData);
            myCurr = myCurr->getNext();
        }
    }
    return *this;
}
```

**Linked List Template Copy Constructor**

```cpp
// Deep copy constructor for LinkListT objects.

template <class Item> LinkListT<Item>::LinkListT(const LinkListT<Item>& Source) {
    Head = Tail = Curr = NULL;
    LinkNodeT<Item>* myCurr = Source.Head; // copy list
    while (myCurr != NULL) {
        Item xferData = myCurr->getData();
        AppendNode(xferData);
        myCurr = myCurr->getNext();
    }
}
```

**Template Functions**

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

```cpp
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:

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

```cpp
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) {
            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).