Chapter 7

Templates

Motivation for Templates

- You want a
  - list of shapes
  - list of frames
- How can you write one *list* class?
  - List all ways you can think of
  - List pros/cons of each method

One Way to Look at Templates…

- Until now, we used variables.
  - The *type* of vars is fixed when you write code.
  - The *value* of vars isn't fixed when you write code.
- With templates, *type* isn't fixed when you write code!
- You can sort of use a *type* as a variable!

Template Example: Queue of some type blah

```
template <class BlahBlah> class Queue {
private:
      BlahBlah buffer[100];
      int head, tail, count;
public:
      Queue();
      void  Insert(BlahBlah item);
      BlahBlah Remove();
      ~Queue();
};
```

Keywords

Parameter, can be any type

Either a class or method definition starts here.

Templates

- Definition of "template":

  Parameterized class with parameters.

  Parameters denote unknown type.
- Usage:

  Situations where *same* algorithms & data structures are applied to *different* data types
- Syntax Example:

```
template <class BlahBlah> class Queue {...}
```


What can a Parameter be used for?

1. as type of local data to class:

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

2. as method argument type:

```
void  Insert(BlahBlah item);
```

3. as method return type:

```
BlahBlah Remove();
```

Instantiating a template

- Given:

```
template <class BlahBlah> class Queue {...}
```

- Instantiate Queue of ints in 2 ways:
  - `Queue<int> intQueue;`
  - `typedef Queue<int> IntegerQueue;`
    `IntegerQueue intQueue;`

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

Usage of Templates

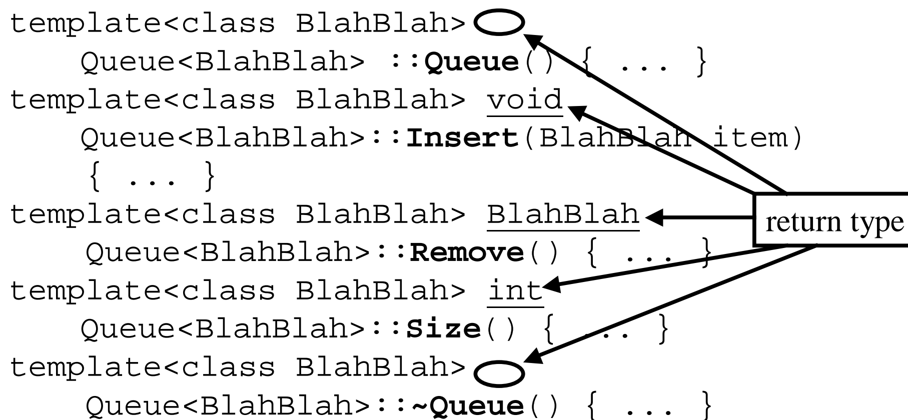
```
intQueue.Insert(100);          // add 100
intQueue.Insert(200);          // add 200
int x = intQueue.Remove();     // remove 100
intQueue.Insert(300);          // queue now
                               // has (200,300)
int x = intQueue.Size();       // size is 2
```

Compiler view of templates…

- Compiler *macro expands* template code:
  - You write `Queue<int> intQueue;`
  - Compiler emits new copy of a class named "Queueint" & substitutes <int> for <blahblah>
  - Therefore, all template code goes in a header file

So
```
   private:
      BlahBlah buffer[100];
```
becomes
```
   private:
      Queueint buffer[100];
```


Syntax for Implementing Template Class Methods



A Complete Template Class

```
template <class BlahBlah> class Queue {
   private:
     BlahBlah buffer[100];
     int head, tail, count;
   public:
   Queue();
     void Insert(BlahBlah item);
     BlahBlah Remove();
     int Size();
     ~Queue();
};
```

All template code goes in a header file

Using the template parameter: arg type, ret type, data.

Ch7 11

A Complete Template Class (Continued)

```
template <class BlahBlah>
   Queue<BlahBlah>::Queue():count(0), head(0),
   tail(0)
{}

template<class BlahBlah> void
   Queue<BlahBlah>::Insert(BlahBlah item) {
      assert(count <100);
      buffer[tail] = item;
      tail = (tail + 1)% 100;
      count++;
}
```

header file

Ch7 12

A Complete Template Class (Continued)

```
template <class BlahBlah> BlahBlah
   Queue<BlahBlah>::Remove() {
      assert(count > 0);
      int val = head;
      head = (head + 1)%100;
      count--;
      return buffer[val];
}
template <class BlahBlah> int
   Queue<BlahBlah>::Size() {return count;}

template <class BlahBlah>
   Queue<BlahBlah> ::~Queue() {}
```

header file

Recap

- Note that method bodies…
  - Use *same* algorithm for queue of ints, Shapes, …
  - But compiler *still* type checks!
    - (It does macro expansion, so if you declare

      `Queue<int>`

      `Queue<char>`

      compiler has two different classes after expansion to use with normal type checking rules.)

3 Implicit Assumptions on Template Parameters: QueueExample

1. Declaration of the array of BlahBlahs:
   Assumes BlahBlah has no-args constructor

```
template <class BlahBlah> class Queue {
    private:
        BlahBlah buffer[100];
         ...
};
```

2. Assignment of BlahBlahs:
   Assumes BlahBlah has appropriately assignment operator.

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


3 Implicit Assumptions on Template Parameters: QueueExample (Contd.)

3. The way that BlahBlahs are returned by Remove method:
   Assumes BlahBlah has appropriate copy constructor.

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

Morale:

- Have fun decoding those compiler error messages!

Variable and Constant Template Parameters

Section 7.3

- Template parameters may be
  - Classes
    - we saw this previously
  - Variables
    - E.g., to specify a size for a data structure
  - Constants
    - Useful to define templates for special cases

Queue Template Class with a Variable Parameter

```
template <class BlahBlah, int size> class Queue {

     private:
        BlahBlah buffer[size];
        int head, tail, count;

     public:
        Queue();
        void      Insert(BlahBlah item);
        BlahBlah Remove();
        int       Size();
                 ~Queue();

};
```


Usage of Template Class with a Variable Parameter

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

Are these legal?

```
smallIntegerQueue = largeIntegerQueue;
smallIntegetQueue = smallRealQueue;
largeIntegerQueue = largeIntegerQueue2;
```

Constant Parameter

- Special cases are sometimes desired

```
class Displayable<int> {
    private:
       int*        displayed;
       TextBox* textBox;
       char* ToString(int v);
       int   FromString(char* text);
    public:
       Displayable(TextBox* tbox);  // textbox to show in
       void ShowThis(int* d);       // what to show
       void Show();                 // show current value
       void Reset();                // reset value from
                                    // textBox
            ~Displayable();
};
```

int does not define ToString or FromString

Special Case Displayable Template (Continued)

```
char* Displayable<int>::ToString(int v) {
          char* buf = new char[10];
          ostrstream format(buf);
          format << v;
          return buf;
}

int Displayable<int>::FromString(char* text) {
          istrstream(text) format;
          int value;
          format >> value;
          return value;
}
```

Section 7.4

Partial Definition of the List Template: A Related Templates Example

```
template <class BaseType> class List {
   private:
         ...
   public:
      List();
      void      First();
      void      Next();
      int       Done();
      BaseType& Current();
      void      Insert(BaseType val);
      void      Delete();
      ~List();
  };
```


The Node Template

```
template <class BaseType> class Node {
     private:
        BaseType value;
        Node<BaseType> *next;

     public:
        Node(BaseType base);
        BaseType& Value();
        void      ConnectTo(Node<BaseType>* nxt);
        Node<BaseType>* Next();
       ~Node();
};
```

Implementation of the Node Template

```
template <class BaseType>
   Node<BaseType>::Node(BaseType base)
     :value(base), next(0) {}
template <class BaseType>
   BaseType& Node<BaseType>::Value(){return value;}

template <class BaseType>
   void Node<BaseType>::ConnectTo(Node<BaseType>* nxt)
   {next = nxt; }
template <class BaseType>
   Node<BaseType>* Node<BaseType>::Next() {return next; }

template <class BaseType>
   Node<BaseType>::~Node(){}
```


Relationship Between the List and Node Templates

```
template <class BaseType> class List {
   private:
      Node<BaseType> *head;     // beginning of list
      Node<BaseType> *current;  // current element
      Node<BaseType> *previous; // previous element;
                                // needed for deletion

    public:
       ...  // shown above
};
```

Implementation of Insert

```
template <class BaseType>
     void List<BaseType>::Insert(BaseType val)
     { Node<BaseType> *newNode = new Node <BaseType> (val);
       if (!head)
          { head = current = newNode;
            return;
          }
       assert(current);
       newNode->ConnectTo(current->Next());
       current->ConnectTo(newNode);
       current = newNode;
}
```


Implementation of Delete

```
template <class BaseType>
    void List<BaseType>::Delete()
    { assert(current);
      Node<BaseType> *temp = current;
      if(current == head)
      { head = head->Next();
        current = head;
        delete temp;
        return;
      }
      assert(previous);
      current = current->Next();
      previous->ConnectTo(current);
      delete temp;
}
```

Section 7.5

Templates and Inheritance

- Template may inherit from Template
- Non-template may inherit from Template
- Template may inherit from non-template
- Templates may use multiple inheritence

Inheritance Between Templates

```
template <class BlahBlah> class Queue {

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

    public:
         Queue();
         void      Insert(BlahBlah item);
         BlahBlah Remove();
                  ~Queue();

};
```

Inheritance Between Templates (cont'd)

Base class is elaborated with same parameterized type

```
template <class BlahBlah>
     class InspectableQueue : public Queue<BlahBlah>
{
  public:
     InspectableQueue();
     BlahBlah Inspect();  // return without removing the
                           //first element
    ~InspectableQueue();
};
```


Using InspectableQueue Template

```
InspectableQueue<Location> locations;
Location loc1(...);
Location loc2(...);
     ...
locations.Insert(loc1);              // base class method
locations.Insert(loc2);              // base class method
     ...
Location front = locations.Remove(); // base class method
Location newFront = locations.Inspect(); // derived class
                                         // method
```

NonTemplate Inheriting from Template

```
template <class BaseType> class List {
  private:
      ...
  public:
     List();
     void       First();
     void       Next();
     int        Done();
     BaseType&  Current();
     void       Insert(BaseType val);
     void       Delete();
               ~List();
};
```


Polygon Class Inherits from List Template

Base class is elaborated with a specific type

```
class Polygon : public List<Location>
                // Polygon "is-a" list of Locations
{
     public:
          Polygon();
          void Draw(Canvas& canvas); // draw itself
};
```

Using Polygon Class

```
 Polygon poly;
      ...
poly.Insert(Location(20,20));    // inherited from template
poly.Insert(Location(30,30));    // inherited from template

// insert other locations

poly.Draw(canvas);               // derived class method
```