Templates

Reminder

- Final exam
  - The date for the Final is **still**:
  - Saturday, November 16th
  - 8am – 10am
  - 01-2000

Announcement

- If you would like a paper copy of Volume 2 of the text, please let me know via e-mail by Wednesday.
  - Topics in Volume 2
    - Exceptions / Assertions
    - STL / Iostreams
    - Runtime Identification
    - Multiple Inheritance
  - Cost will be $25.

Before we begin

- Project Notes
  - Parking Lot Problem
    - All cars have a width of 1
    - No 1x1 cars
  - Clock problem due Oct 16.
    - Wednesday (midnight)

Exam 1

```
Customer::Customer (string name, string
  address, Currency startingBalance) : myName
  (name), myAddr (address), myAcct
  (Account(name,startingBalance))
{}
```

Is okay

Plan

- Today: Templates
- Thursday: STL 1
- Monday: STL 2
- Tuesday: exam 2
- Thursday: IOStreams 1

Exam 2

- October 22 (Week from today)
- Topics
  - Memory Management / Pointers
  - Inheritance / Abstract Classes
  - Templates (use of)
  - Testing

Questions

A quick intro to Templates

- Problem:
  - Let's say that we need a Queue class that manages a Queue of ints:

```
class intQueue
{
 private:
    int *q;
    int n;
    …
public:
    void enqueue (int i);
    int dequeue();
   …
}
```

A quick intro to Templates

- Problem:
  - Now, let's say that we need a Queue class that manages a Queue of double:

```
class doubleQueue
{
 private:
    double *q;
    int n;
    …
public:
    void enqueue (double i);
    double dequeue();
   …
}
```

A quick intro to Templates

- Note that the code for intQueue will be almost identical to that of doubleQueue.
- Is there a way to reuse the basic Queue code but have it work on different datatypes?

A quick intro to Templates

- The Java solution

```
class objectQueue
{
 private:
    Object *q;
    int n;
    …
public:
    void enqueue (Object i);
    Object dequeue();
   …
}
```

A quick intro to Templates

- The C++ implementation of the Java solution

```
class voidQueue
{
 private:
    void **q;
    int n;
    …
public:
    void enqueue (void *i);
    void * dequeue();
   …
}
```

A quick intro to Templates

- Problems with this approach

```
int *i = new int (9);  // You must use pointers
float *f = new float (5.7);

voidQueue Q;
Q.enqueue (i);
Q.enqueue (f);  // Multiple datatypes in same
  queue

int *g = (int *)(Q.dequeue);// must cast, is this
                            // an int * or a
  float *
```

A quick intro to Templates

- The template solution:
  - Define a "generic" queue class
  - Datatype of the objects managed by the queue is defined later
  - Can define such a class using Templates:

A quick intro to Templates

```
template <class T>
class Queue
{
 private:
    T *q;
    int n;
    …
public:
    void enqueue (T i);
    T dequeue();
   …
}
```

Datatype to be filled in later

A quick intro to Templates

- To use this template:

```
// a queue of ints
Queue<int> iqueue;

// a queue of doubles
Queue<double> dqueue;

// a queue of aClass objects
Queue<aClass> aqueue

// a queue of pointers to aClass
Queue<aClass *> aptrqueue
```

Template Example

- Safe array
  - Looks and feels like an array
  - Will be responsible for it's own memory management
  - Will fail if you try to access beyond the bounds of the array
  - Can hold any datatype or object.

Template Example

```
template < class T >
class SafeArray {
public:
   SafeArray( int initSize );
  ~SafeArray();

  T &operator[]( int index );
  int getSize();
private:
  T *data; // an array
  int size;
}
```

Template Example

```
// constructor
template < class T >
SafeArray< T >::SafeArray( int initSize ):
  data( 0 ), size( initSize )
{
  assert (initsize > 0)
  data = new T[ size ];
}

// destructor
template < class T >
SafeArray< T >::~SafeArray()
{
    delete[] data;
}
```

Template Example

- operator[] returns a reference to T

```
// operator[]
//
template < class T >
T &SafeArray< T >::operator[]( int index )
{
     assert( ( index >= 0 ) && ( index < size )
  );
     return data[ index ];
}
```

Template Example

```
int N = 5;
SafeArray< int > i_good( N );
SafeArray< std::string > s_good( N );
s_good[0] = "bridge";
s_good[1] = "to";
s_good[2] = "the";
s_good[3] = "21st";
s_good[4] = "century";

for ( int x = 0 ; x < N; x += 1 )
{
     i_good[ x ] = 2 * x;
}

i_good[5] = 24;  // Will cause assertion to fail.
```

Alternate implementation of operator[]

```
template < class T >
T &SafeArray< T >::operator[]( int index )
{
   if (index >= size) {
     // create a new array capable of holding the indexed element
     T *new_data (new T[ index + 1 ]);

     // copy the data over from the old to new array
     for (int i( 0 ); i < size; i++) { new_data[ i ] = data[ i ]; }

     // delete the old array
     delete[] data;

     // point data to the new array
     data = new_data;
     // increase the size count
     size = index + 1;
   }
return data[ index ];
}
```

Template arguments

- Template arguments can be of ordinary types:

```
template < class T, int S >
class SafeArray {
public:
   SafeArray();
  ~SafeArray();

  T &operator[]( int index );
  int getSize();
private:
   T *data; // an array
  int size;
}
```

Template arguments

- Template arguments can be of ordinary types:

```
// constructor
template < class T, int S >
SafeArray< T, S >::SafeArray( ):
  data( 0 ), size( S )
{
  assert (S > 0)
  data = new T[ size ];
}
```

Template arguments

- Template arguments can be of ordinary types:

```
SafeArray< int, 5 > i_good;

for ( int x = 0 ; x < N; x += 1 )
{
     i_good[ x ] = 2 * x;
}

i_good[5] = 24;  // Will cause assertion to fail.
```

Template arguments

- Template arguments can have defaults

```
template < class T, int S=10>
class SafeArray {
public:
   SafeArray();
  ~SafeArray();

  T &operator[]( int index );
  int getSize();
private:
   T *data; // an array
  int size;
}
```

Template arguments

```
SafeArray< int > i_good;
```

Will be an int smart array of size 10

Template arguments

- Template arguments can be given in terms of other template arguments

```
template < class T, T default_val >
class SafeArray {
public:
   SafeArray(int initize);
  ~SafeArray();

  T &operator[]( int index );
  int getSize();
private:
   T *data; // an array
  int size;
}
```

Template arguments

- Template arguments can be given in terms of other template arguments

```
// constructor
template < class T, T default_val >
SafeArray< T, default_val >::SafeArray( int initSize ):
  data( 0 ), size( initsize )
{
  assert (size > 0)
  data = new T[ size ];
  for (int i=0; i < size; i++) data[i] = default_val;
}
```

Template arguments

- Template arguments can be given in terms of other template arguments

```
SafeArray<float, 3.4> farray;
SafeArray<int, 12> iarray;
SafeArray<string, "foo"> sarray;
```

Class Templates

- Questions?

Function Templates

- Functions can also be templated
  - Data type of function arguments and return type filled in later.
  - Good for "common" algorithms
    - Search
    - Sort
    - Generic functions

Function Templates

```
// max
// returns the maximum of the two
  elements
template <class T>
T max(T a, T b)
{
    return a > b ? a : b ;
}
```

Function Templates

- Use a template function like a regular function
  - No need to specify type explicitly

```
void main()
{
   cout << "max(10, 15) = " << max(10, 15) << endl ;
   cout << "max('k', 's') = " << max('k', 's') << endl ;
   cout << "max(10.1, 15.2) = " << max(10.1, 15.2) <<
           endl ;
 }
```

Program Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
```

Function Templates

- However, object passed to max must define the > operation.

```
main (int argc, char *argv[])
{
   Foo f1 (1,2);
   Foo f2 (2,3);
   Foo f3 (4,4);
   f3 = mymax (f1, f2);
}
```

```
"foo.cpp", line 10: Error: The operation "Foo > Foo" is illegal.
"foo.cpp", line 25:     Where: While instantiating "mymax<Foo>(Foo,
   Foo)".
"foo.cpp", line 25:     Where: Instantiated from non-template code.
1 Error(s) detected.
```

Templates

- Note that compiler will actually generate functions/classes for each instantiated templated class or function it encounters.
- When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.
  - A class generated from a class template is called a generated class.
  - A function generated from a function template is called a generated function.

Templates

- Templates get resolved at compile-time.
  - Unlike polymorphism that gets resolved at runtime.

Summary

- Templates
  - Generic Programming
  - Datatypes filled in later
  - Class Templates
  - Function Templates
  - Next time: The Standard Template Library
- Questions?