CMSC 341

C++ and OOP

What you should already know

- Basic C++ class syntax
- Default parameters and parameter passing
- Initializer list
- Proper use of const
- vector and string classes
- Big 3
- Pointers and dynamic memory management
- Templates

Intcell.H

```
#ifndef IntCell_H
#define IntCell_H

// A class for simulating an integer memory cell.
class IntCell
 {
    public:
          explicit IntCell( int initialValue = 0 );
          IntCell( const Intcell & ic );
          ~IntCell( );
          const IntCell & operator =( const IntCell & rhs );

          int Read( ) const;
          void Write( int x );

    private:
          int m_storedValue;
};

#endif
```

IntCell.cpp (part 1)

```
#include "IntCell.h”
using namespace std;

// Construct the IntCell with initialValue
IntCell::IntCell( int initialValue ) :
   m_storedValue( initialValue )
{
   // no code
}

//copy constructor
IntCell::IntCell( const IntCell & ic )
{
   Write ( ic.Read( ) );
}


// destructor
IntCell::~IntCell( )
{
   // no code
}
```

IntCell.cpp (part 2)

```
//assignment operator
const IntCell & IntCell::operator=( const IntCell & rhs )
{
   if (this != &rhs)
       Write( rhs.Read( ) );

   return *this;

}

// Return the stored value (accessor)
int IntCell::Read(  ) const
{
   return m_storedValue;
}

// Store x (mutator)
void IntCell::Write( int x )
{
   m_storedValue = x;
}
```

TestIntCell.C

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

int main( )
    {
    IntCell m;   // Or, IntCell m( 0 ); but not IntCell m( );
    IntCell n;

    n = m;
    m.Write( 5 );
    cout << "Cell m contents: " << m.Read( ) << endl;
    cout << "Cell n contents: " << n.Read( ) << endl;

    return 0;
}
```

Function Templates

- A pattern for a function that has a type-independent algorithm
- Not a function itself
- Parameteric polymorphism through the template parameter
- Not compiled until type is known

```cpp
//
// Return the maximum item in array a.
// Assumes a.size( ) > 0.
// “Comparable” objects must provide
//  operator< and operator=

template <typename Comparable>
const Comparable &
findMax( const vector<Comparable> & a )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )
        if( a[ maxIndex ] < a[ i ] )
            maxIndex = i;

    return a[ maxIndex ];
}
```

```
//Example code using function template “findMax”

int main( )
{
    vector<int>      v1( 37 );
    vector<double>  v2( 40 );
    vector<string>  v3( 80 );
    vector<IntCell> v4( 75 );

    // Additional code to fill in the vectors not shown

    cout << findMax( v1 ) << endl;  // OK: Comparable = int
    cout << findMax( v2 ) << endl;  // OK: Comparable = double
    cout << findMax( v3 ) << endl;  // OK: Comparable = string
    cout << findMax( v4 ) << endl;  // Illegal; operator< undefined

    return 0;
}
```

Class Templates

- A cookie cutter for a class – NOT a class itself
- Parameteric polymorphism
- Type-independent classes
- Implementation is in the header file
- Not compilable
- Object vs. Comparable template parameter

```cpp
// MemCell.h (part 1)

#ifndef MEMCELL_H
#define MEMCELL_H

// A class for simulating a memory cell.
template <class Object>
class MemCell
{
  public:
      explicit MemCell(const Object &initialValue = Object( ) );
      MemCell(const MemCell & mc);

      const MemCell & operator= (const MemCell & rhs);
      ~MemCell( );

      const Object & Read( ) const;
      void Write( const Object & x );

  private:
      Object m_storedValue;
};
// MemCell implementation follows
// MemCell.h(part 2)

// Construct the MemCell with initialValue
template <class Object>
MemCell<Object>::MemCell( const Object & initialValue )
                  :m_storedValue( initialValue )
{
   // no code
}

//copy constructor
template <class Object>
MemCell<Object>::MemCell(const MemCell<Object> & mc)
{
   Write( mc.Read( ) );
}
//assignment operator
template <class Object>
const MemCell<Object> &
    MemCell<Object>::operator=(const MemCell<Object> & rhs)
{
   if (this != &rhs) Write( rhs.Read(  ) );
   return *this;
}
```

```
// MemCell.h (part 3)
// destructor
template <class Object>
     MemCell<Object>::~MemCell( )
{
   // no code
}

// Return the stored value.
template <class Object>
const Object & MemCell<Object>::Read( ) const
{
   return m_storedValue;
}

// Store x.
template <class Object>
void MemCell<Object>::Write( const Object & x )
{
   m_storedValue = x;
}

#endif // end of MemCell.h
```

TestMemCell.C

```
#include <iostream>
#include <string>
#include "MemCell.h"
using namespace std;

int main( )
 {
   MemCell<int>    m1;
   MemCell<string> m2( "hello" );

   m1.Write( 37 );
   string str = m2.Read();
   str += " world";
   m2.Write(str);

   cout << m1.Read( ) << endl << m2.Read( ) << endl;

   return ( 0 );
}
```

Implementing Templates

When compiling code that instantiates a class template, the compiler must have both the class definition and the class implementation available. There are two ways to accomplish this.

1. As in CMSC 202, the template definition is placed in XX.h and the implementation is placed in XX.cpp which was then #included at the bottom of XX.h
2. More customary, and allowed in this class, is to simply write the implementation code inside XX.h after the class definition.

Compiling Templates

While it is possible to compile a template file, creating a .gch file, there is no need to do so.

Recall that template code is compiled when the template is instantiated.

Compiling with -I

In some projects in this course you will be give .h files which you are to use without modification. These files will be located in some publicly accessible directory, say

/afs/umbc.edu/users/f/r/frey/pub/CMSC341

To use .h files from this directory, your makefile must include the following

1. DIR=/afs/umbc.edu/users/f/r/frey/pub/CMSC341
   which defines the symbol DIR (like a #define in C)
2. CCFLAGS= -ansi –Wall –I . –I $(DIR)
   which defines the compiler flags to be used.
   In particular, -I $(DIR) tells the compiler to look in DIR for .h files that it can’t find in the “usual” places