



# C++ Review

---

CptS 223 – Advanced Data Structures

Larry Holder

School of Electrical Engineering and Computer Science  
Washington State University



# Purpose of Review

---

- Review some basic C++
- Familiarize us with Weiss's style
- Introduce specific constructs useful for implementing data structures



# Class

---

- The Class defines the data structure and the operations that access and manipulate it
  - Member data
  - Member functions or methods
- Encapsulation = data + methods
- Information hiding
  - Public vs. private

```
1  /**
2   * A class for simulating an integer memory cell.
3   */
4  class IntCell
5  {
6   public:
7   /**
8   * Construct the IntCell.
9   * Initial value is 0.
10  */
11  IntCell( )
12   { storedValue = 0; }
13
14  /**
15  * Construct the IntCell.
16  * Initial value is initialValue.
17  */
18  IntCell( int initialValue )
19   { storedValue = initialValue; }
```

Comment

Constructor

Constructor

```
21     /**
22      * Return the stored value.
23      */
24     int read( )
25         { return storedValue; }
26
27     /**
28      * Change the stored value to x.
29      */
30     void write( int x )
31         { storedValue = x; }
32
33     private:
34         int storedValue;
35 };
```

Information  
Hiding

Encapsulation



# Extra Syntax

---

- Default parameters
- Initializer list
- Explicit constructor
- Constant member function
  - Accessor methods
  - Mutator methods

```
1  /**
2   * A class for simulating an integer memory cell.
3   */
4  class IntCell
5  {
6  public:
7      explicit IntCell( int initialValue = 0 )
8          : storedValue( initialValue ) { }
9      int read( ) const
10         { return storedValue; }
11     void write( int x )
12         { storedValue = x; }
13
14     private:
15         int storedValue;
16 };
```

Explicit  
Constructor

Default  
Parameters

Accessor

Initializer  
List

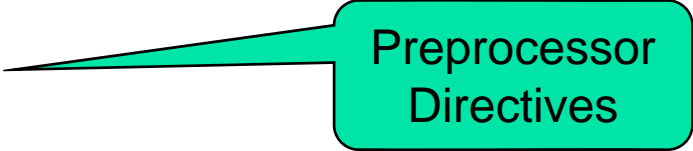
```
IntCell obj;
obj = 37;
```



# Separation of Interface and Implementation

---

- Interface (.h) file
  - Defines class and its member data and functions
- Implementation (.cc or .cpp) file
  - Provides implementations of member functions



Preprocessor  
Directives

```
1  #ifndef IntCell_H
2  #define IntCell_H
3
4  /**
5   * A class for simulating an integer memory cell.
6   */
7  class IntCell
8  {
9      public:
10     explicit IntCell( int initialValue = 0 );
11     int read( ) const;
12     void write( int x );
13
14     private:
15     int storedValue;
16 };
17
18 #endif
```

IntCell class interface in file *IntCell.h*.

```
1 #include "IntCell.h"
2
3 /**
4  * Construct the IntCell with initialValue
5  */
6 IntCell::IntCell( int initialValue ) : storedValue( initialValue )
7 {
8 }
9
10 /**
11  * Return the stored value.
12  */
13 int IntCell::read( ) const
14 {
15     return storedValue;
16 }
17
18 /**
19  * Store x.
20  */
21 void IntCell::write( int x )
22 {
23     storedValue = x;
24 }
```

Preprocessor  
Directive

Scoping  
Operator  
ClassName::member

IntCell class implementation in file *IntCell.cpp*.

```
1  #include <iostream>
2  #include "IntCell.h"
3  using namespace std;
4
5  int main( )
6  {
7      IntCell m;    // Or, IntCell m( 0 ); but not IntCell m( );
8
9      m.write( 5 );
10     cout << "Cell contents: " << m.read( ) << endl;
11
12     return 0;
13 }
```

Preprocessor  
Directives

Default class

Program using IntCell class in file *TestIntCell.cpp*.



# C++ Details

---

- Pointers
- Parameter passing
- Return passing
- Reference variables
- Destructor, copy constructor, operator=



# Pointers

---

```
1  int main( )
2  {
3      IntCell *m;
4
5      m = new IntCell( 0 );
6      m->write( 5 );
7      cout << "Cell contents: " << m->read( ) << endl;
8
9      delete m;
10     return 0;
11 }
```

Address-of operator: &

```
IntCell icObj;
IntCell *m = & icObj;
```

No automatic garbage  
collection in C++



# Parameter Passing

---

- Call by value
  - Small objects not altered by function
- Call by constant reference
  - Large objects not altered by function
- Call by reference
  - Objects altered by function

```
double avg (const vector<int> & arr, int n, bool & errorFlag);
```

```
1  const string & findMax( const vector<string> & arr )
2  {
3      int maxIndex = 0;
4
5      for( int i = 1; i < arr.size( ); i++ )
6          if( arr[ maxIndex ] < arr[ i ] )
7              maxIndex = i;
8
9      return arr[ maxIndex ];
10 }
11
12 const string & findMaxWrong( const vector<string> & arr )
13 {
14     string maxValue = arr[ 0 ];
15
16     for( int i = 1; i < arr.size( ); i++ )
17         if( maxValue < arr[ i ] )
18             maxValue = arr[ i ];
19
20     return maxValue;
21 }
```

### Return Passing

Make sure the object returned will persist after returning from the function call.



# Reference Variables

---

- As seen, can be used for parameter passing
- Also used as synonyms for the objects they reference
  - Avoid cost of copying

```
string x = findMax (a);  
cout << x << endl;
```

```
const string & x = findMax (a);  
cout << x << endl;
```



# Destructor, Copy Constructor, operator=

---

- Default definitions for all classes
- Destructor
  - Called when object goes out of scope or subject to a **delete**
  - By default, calls destructor on all data members
  - Can use to **delete** objects created using **new**
  - Can use to close any opened files.



# Destructor, Copy Constructor, operator=

---

- Copy constructor
  - Declaration with initialization
    - `IntCell B = C;`
    - `IntCell B (C);`
  - Object passed using call by value (instead of by `&` or `const &`)
  - Object returned by value (instead of by `&` or `const &`)
- Simple assignment for all members with primitive data types (e.g., `int`, `double`, ...)
- Calls copy constructors on all member objects



# Destructor, Copy Constructor, operator=

---

- Copy assignment operator: **operator=**
- Called when objects on both sides of assignment already constructed
  - E.g., 

```
IntCell B(0);  
IntCell C(1);  
B = C;
```
- By default, operator= called on each data member of objects

```
1  IntCell::~IntCell( )
2  {
3      // Does nothing, since IntCell contains only an int data
4      // member. If IntCell contained any class objects, their
5      // destructors would be called.
6  }
7
8  IntCell::IntCell( const IntCell & rhs ) : storedValue( rhs.storedValue )
9  {
10 }
11
12 const IntCell & IntCell::operator=( const IntCell & rhs )
13 {
14     if( this != &rhs ) // Standard alias test
15         storedValue = rhs.storedValue;
16     return *this;
17 }
```

Default destructor, copy constructor and operator= for IntCell



# Problems with Defaults

---

```
1  class IntCell
2  {
3      public:
4          explicit IntCell( int initialValue = 0 )
5              { storedValue = new int( initialValue ); }
6
7          int read( ) const
8              { return *storedValue; }
9          void write( int x )
10             { *storedValue = x; }
11     private:
12         int *storedValue;
13 };
```



# Problems with Defaults

---

```
1  int f( )
2  {
3      IntCell a( 2 );
4      IntCell b = a;
5      IntCell c;
6
7      c = b;
8      a.write( 4 );
9      cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;
10     return 0;
11 }
```

Output?



# Fixing the Defaults

---

```
21  IntCell::IntCell( const IntCell & rhs )
22  {
23      storedValue = new int( *rhs.storedValue );
24  }
25
26  IntCell::~~IntCell( )
27  {
28      delete storedValue;
29  }
30
31  const IntCell & IntCell::operator=( const IntCell & rhs )
32  {
33      if( this != &rhs )
34          *storedValue = *rhs.storedValue;
35      return *this;
36  }
```



# Templates

---

- Designing type-independent data structures and algorithms
- Function templates
- Class templates



# Function Templates

---

```
1  /**
2   * Return the maximum item in array a.
3   * Assumes a.size( ) > 0.
4   * Comparable objects must provide operator< and operator=
5   */
6  template <typename Comparable>
7  const Comparable & findMax( const vector<Comparable> & a )
8  {
9      int maxIndex = 0;
10
11     for( int i = 1; i < a.size( ); i++ )
12         if( a[ maxIndex ] < a[ i ] )
13             maxIndex = i;
14     return a[ maxIndex ];
15 }
```



# Function Templates

---

```
1  int main( )
2  {
3      vector<int>    v1( 37 );
4      vector<double> v2( 40 );
5      vector<string> v3( 80 );
6      vector<IntCell> v4( 75 );
7
8      // Additional code to fill in the vectors not shown
9
10     cout << findMax( v1 ) << endl; // OK: Comparable = int
11     cout << findMax( v2 ) << endl; // OK: Comparable = double
12     cout << findMax( v3 ) << endl; // OK: Comparable = string
13     cout << findMax( v4 ) << endl; // Illegal; operator< undefined
14
15     return 0;
16 }
```



# Operator Overloading

---

- Define the meaning of a built-in operator

```
class IntCell
{
public:
    bool operator< (const IntCell & rhs) const
        { return storedValue < rhs.storedValue; }

    void print (ostream & out) const
        { out << " IntCell(" << storedValue << ")"; }

    ...
}

ostream & operator<< (ostream & out, const IntCell & rhs)
{ rhs.print (out); return out; }
```



# Class Templates

---

```
1  /**
2   * A class for simulating a memory cell.
3   */
4  template <typename Object>
5  class MemoryCell
6  {
7      public:
8          explicit MemoryCell( const Object & initialValue = Object( ) )
9              : storedValue( initialValue ) { }
10         const Object & read( ) const
11             { return storedValue; }
12         void write( const Object & x )
13             { storedValue = x; }
14     private:
15         Object storedValue;
16 };
```

Zero may not be a  
valid Object



# Class Templates

---

```
1  int main( )
2  {
3      MemoryCell<int>    m1;
4      MemoryCell<string> m2 ( "hello" );
5
6      m1.write( 37 );
7      m2.write( m2.read( ) + "world" );
8      cout << m1.read( ) << endl << m2.read( ) << endl;
9
10     return 0;
11 }
```



# Summary

---

- Basic C++
- Templates
- Tools for easing the design of type-independent data structures and algorithms