WASHINGTON STATE UNIVERSITY

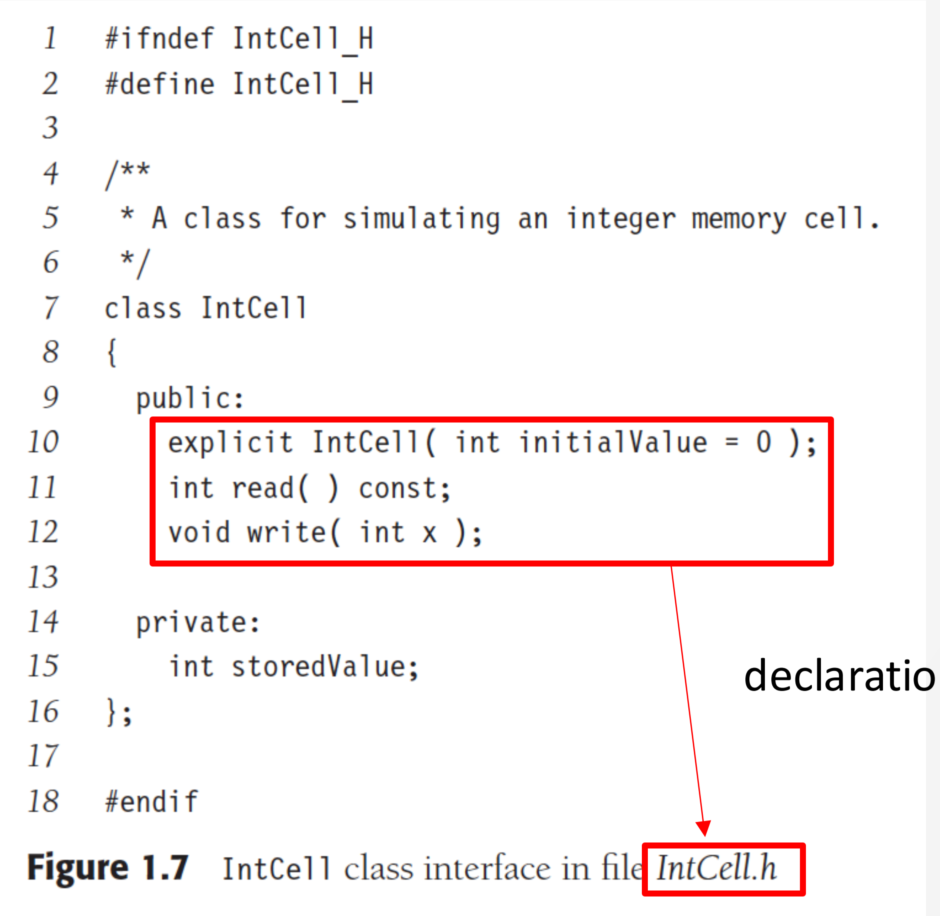
(1-2) C++ Higher Features

C++ Features

C++ features

- Separation of interface and implementation (a design principle)
- Lvalues, Rvalues, and References
- std::swap & std::move
- Big Five: why and when do you need to do them?
  - Destructor
  - Copy constructor
  - Move constructor (introduced in C++11)
  - Copy Assignment operator=
  - Move Assignment operator= (introduced in C++11)

Separation strategy



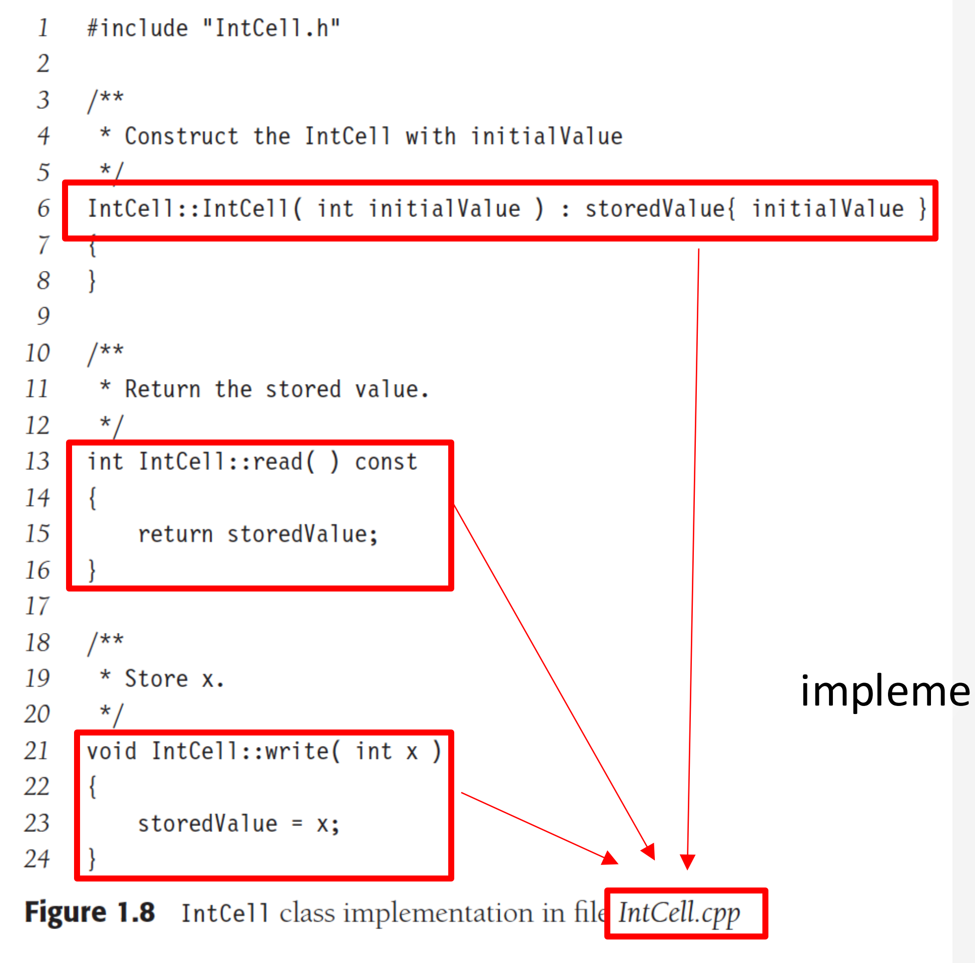
```
#ifndef IntCell_H
#define IntCell_H

/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );
    int read( ) const;
    void write( int x );

  private:
    int storedValue;
};

#endif
```

Figure 1.7 IntCell class interface in file *IntCell.h*

declaration in header files (.h)

Example on Page 16, textbook

Separation strategy



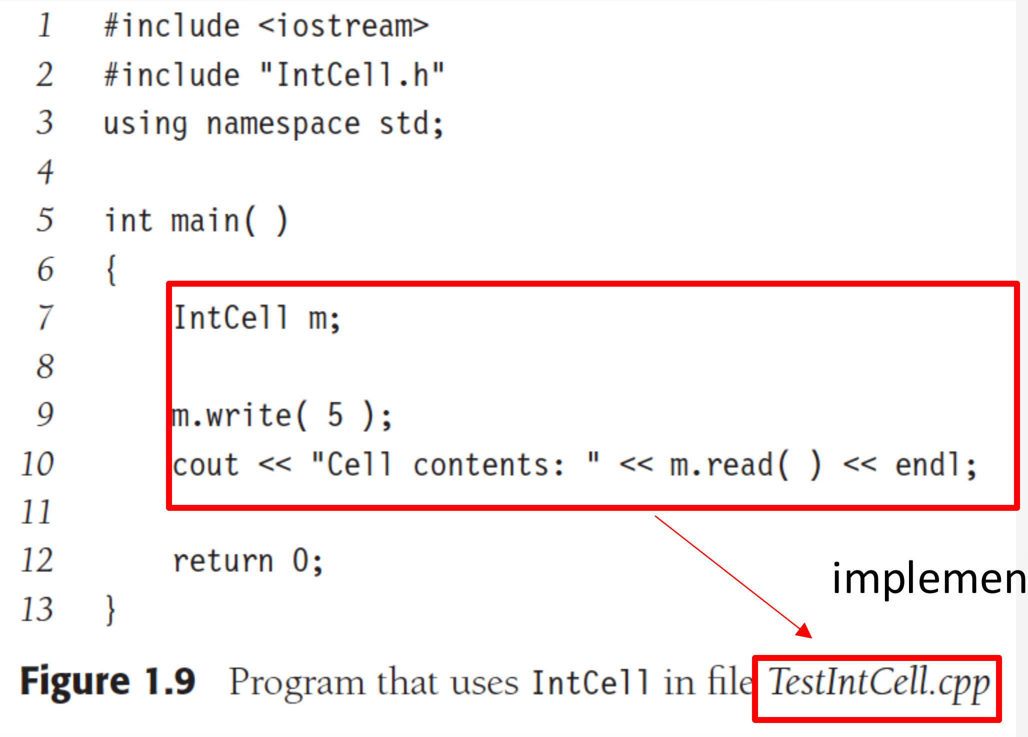
```
#include "IntCell.h"

/**
 * Construct the IntCell with initialValue
 */
IntCell::IntCell( int initialValue ) : storedValue{ initialValue }
{
}

/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}

/**
 * Store x.
 */
void IntCell::write( int x )
{
    storedValue = x;
}
```

Figure 1.8 IntCell class implementation in file *IntCell.cpp*

implementation in source files (.cpp)

Example on Page 17, textbook

Separation strategy



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

int main( )
{
    IntCell m;

    m.write( 5 );
    cout << "Cell contents: " << m.read( ) << endl;

    return 0;
}
```

Figure 1.9 Program that uses IntCell in file *TestIntCell.cpp*

implementation in source files (.cpp)

Example on Page 17, textbook

C++ templated class

```
template <class T>
class MyPair {
          T values [2];
  public:
          mypair (T first, T second)
          {

          values[0]=first;
          values[1]=second;

     }
};
```

- Templated class
- Type of ‘T’ defined at instantiation time
- Can have multiple templated types
- Works well if ‘T’ follows good OO design

```
MyPair<int> pair1 = new MyPair<int>();
```

Lvalues and Rvalues

- Lvalues
  - Permanent variables or objects
  - Persist beyond immediate use
  - Passed to a function with &
    - string & rstr = str;
- Rvalues
  - Temporary values
  - Could be as simple as a number
    - myfunc(2)
    - ‘2’ is a Rvalue

How about x+y? A lvalue or rvalue?

std::swap & std::move

- std::swap ->
  - Swaps two elements via lvalue refs
- std::move ->
  - Moves from src to target using rvalue
  - Effectively “steals” contents of old object for the values in the new object

std::swap & std::move

```
template <class T> swap(T& a, T& b) {
    T tmp(a); // we now have two copies of a
    a = b; // we now have two copies of b (+ discarded a copy of a)
    b = tmp; // we now have two copies of tmp (+ discarded a copy of b)  }
```

```
template <class T> swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);  }
```

The Big-Five

- Big Five: why and when do you need to do them? Recourse management
  - Copy constructor
  - Move constructor
  - Copy Assignment operator=
  - Move Assignment operator=
  - Destructor

Interface of the Big-Five

```
~IntCell( );                                     // Destructor
IntCell( const IntCell & rhs );                  // Copy constructor
IntCell( IntCell && rhs );                       // Move constructor
IntCell & operator= ( const IntCell & rhs );    // Copy assignment operator
IntCell & operator= ( IntCell && rhs );          // Move assignment operator
```

Copy and Move Constructor

```
IntCell B (C);
// Copy construct if C is lvalue (the compiler follows the rule)
// Move construct if C is rvalue
// the compiler follows the C++ rule
```

Copy and Move Assignment

```
lhs = rhs
// copy if rhs is a lvalue, move if rhs is a rvalue

// Copy: IntCell B = A

// Move: IntCell B = new IntCell()
```

Shallow vs deep copies

- A *shallow copy* of an **object** copies all of the member field values
    - Will work if the fields are values
    - Not work if fields are pointers to memory
- A *deep copy* copies all fields, *and* makes copies of dynamically allocated memory pointed to by the fields

Destructor

- C++ compiler will create a default one if you don’t have one
- When do we need to implement a destructor?
  - If you have dynamically allocated memory (create by “new”)