

Review: Objects and classes

(Chapter 2)

CS 3358
Spring 2015

Jill Seaman

Sections 2.1, 2.2, 2.3, 2.8

1

Object Oriented Programming

- An object contains
 - data (or “state”)
 - functions that operate over its data
- Usually set up so code outside the object can access the data only via the member functions.
- If the representation of the data in the object needs to change:
 - The object’s functions must be redefined to handle the changes.
 - The code outside the object does not need to change, it accesses the object in the same way.

2

Object Oriented Programming Concepts

- **Encapsulation:** combining data and code into a single object.
- **Information hiding** is the ability to hide the details of data representation from the code outside of the object.
- **Interface:** the mechanism that code outside the object uses to interact with the object.
 - The prototypes/signatures of the object’s public functions.

3

The Class

- A class in C++ is similar to a structure.
- A class contains:
 - variables (members) AND
 - functions (member functions or methods)
- Members can be:
 - private: inaccessible outside the class (this is the default)
 - public: accessible outside the class.

4

Example class: IntCell

```
class IntCell
{
public:
    // Construct an IntCell. Initial value is 0
    IntCell ()
    { storedValue = 0; }

    // Construct an IntCell. Initial value is initialValue
    IntCell (int initialValue)
    { storedValue = initialValue; }

    // Return the stored value.
    int read ()
    { return storedValue; }

    // Change the stored value to x.
    void write (int x)
    { storedValue = x; }

private:
    int storedValue;
};
```

How is this definition different from the way you defined classes in your previous course?

5

IntCell class

- one data member, four member functions
- private members:
 - storedValue: not visible outside the class
- public members:
 - the four member functions
 - visible and accessible to any function
- constructors
 - describes how instances are created
 - if none, a default constructor is supplied

6

Using IntCell

```
int main()
{
    IntCell m; // calls IntCell() constructor

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

    return 0;
};
```

To compile + run:
put IntCell declaration and this main function in one file (IntCell.cpp),
add #include <iostream> using namespace std;
...\$ g++ IntCell.cpp
...\$./a.out

Output:

```
Cell contents: 5
```

7

IntCell, version 2

```
class IntCell
{
public:
    explicit IntCell (int initialValue = 0)
    : storedValue (initialValue)
    { }

    int read () const
    { return storedValue; }

    void write (int x)
    { storedValue = x; }

private:
    int storedValue;
};
```

What is different from version 1 (other than not having comments)?

8

Four changes to IntCell

1. Default parameter:

```
IntCell (int initialValue = 0)
```

- This constructor has an optional parameter. If no argument is specified, initialValue will be 0.

```
IntCell x;  
IntCell y(5);
```

2. Constant member function

```
int read () const
```

- const after param-list declares function will not change any member values
- signifies function is an accessor (not a mutator)

Four changes to IntCell

3. Initializer list : storedValue (initialValue)

- placed before the constructor body,
- assigns the value of the parameter initialValue to storedValue.
- if storedValue is an object, uses its one-argument constructor to initialize storedValue.

4. "explicit" constructor

- applies to one-argument constructors only
- Prevents compiler from doing this implicit conversion:

```
IntCell obj;  
obj = 37; //should be an error
```

```
IntCell obj;  
IntCell temp(37);  
obj = temp;
```

Separation of Interface from Implementation

- Interface: "What"
 - Class declarations with data members and function prototypes only
 - stored in their own header files (*.h)
- Implementation: "How"
 - Member function definitions are stored in a separate file (*.cpp) Requires use of the scope resolution operator ::
 - must #include the corresponding header file
- Any file using the class should #include *.h
- *.cpp can change without recompiling its users

IntCell, version 3

IntCell.h:

```
#ifndef _IntCell_H_  
#define _IntCell_H_  
  
class IntCell  
{  
public:  
    IntCell (int initialValue = 0);  
    int read () const;  
    void write (int x);  
  
private:  
    int storedValue;  
};  
#endif
```

Note the "include guards" which prevent the file from being included more than once

IntCell, version 3

IntCell.cpp:

```
#include "IntCell.h"

IntCell::IntCell (int initialValue)
: storedValue (initialValue)
{ }

int IntCell::read () const
{
    return storedValue;
}

void IntCell::write (int x)
{
    storedValue = x;
}
```

Function signatures must match exactly with class declaration, but default params are not required

Note the scope resolution operations: **IntCell::**
Indicates which class the function is a member of

13

Using IntCell, version 3

IntCellDriver.cpp:

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

int main()
{
    IntCell m; // calls IntCell() constructor

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

    return 0;
};
```

Do NOT include IntCell.cpp!

To compile and run:
...\$ g++ IntCell.cpp IntCellDriver.cpp
...\$./a.out

14

The Big Three destructor, copy constructor, operator=

- these functions are provided by default, but the default behavior may or may not be appropriate.
- **Destructor**
 - called when object is destroyed (goes out of scope or deleted (explicitly deallocated))
 - responsible for freeing resources used by object
 - ➔ calling delete on dynamically allocated objects
 - ➔ closing files
 - default destructor applies destructor to each member

The Big Three destructor, copy constructor, operator=

- **Copy Constructor**
 - ❖ special constructor, constructs new object from an existing one
 - ❖ called:
 - ➔ for a declaration with initialization:

```
IntCell obj = otherObj;
IntCell obj(otherObj);
```

But not for:
obj = otherObj
 - ➔ when object is passed by value
 - ➔ when object is returned (by value).
 - ❖ default copy constructor:
 - ➔ uses assignment for primitive-type data members
 - ➔ uses copy constructor for object-type data members

16

The Big Three destructor, copy constructor, operator=

- **operator=**

- called when = operator is used on existing objects:

```
obj = otherObj;
```

- default operator= applies = to each member
(aka member-wise assignment)

17

The Big Three destructor, copy constructor, operator=

- When do the defaults not work?
- Generally, when one of the members is dynamically allocated by the class (via a pointer).
- As an example, let's rewrite IntCell and store the value in a dynamically allocated memory location.

18

IntCell, version 4

```
class IntCell
{
public:
    IntCell (int initialValue = 0);
    int read () const;
    void write (int x);

private:
    int *storedValue;
};
```

```
IntCell::IntCell (int initialValue)
{ storedValue = new int;
  *storedValue = initialValue; }
```

```
int IntCell::read () const
{ return *storedValue; }
```

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

What is different from
version 3?

19

IntCell, v. 4, problem with defaults

```
int main()
{
    IntCell a(2);
    IntCell b = a;      //copy constructor
    IntCell c;

    c = b;              //operator=

    a.write(4);

    cout << a.read() << endl
         << b.read() << endl
         << c.read() << endl;
};
```

What is output?

```
4
2
2
```

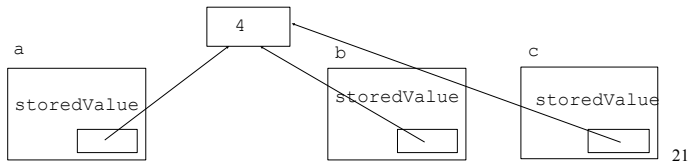
or

```
4
4
4
```

20

IntCell, v. 4, problem with defaults

- Why are they all changed to 4?
- Default copy constructor and operator= all do a **shallow copy**. They copy the pointer instead of what the pointer points to.
- As an result, all 3 objects point to the same location in memory



IntCell, version 5

```
class IntCell
{
public:
    IntCell (int initialValue = 0);

    IntCell(const IntCell &rhs);
    ~IntCell();
    void operator= (const IntCell & rhs);

    int read () const;
    void write (int x);

private:
    int *storedValue;
};
```

Note the prototypes for the big 3

22

IntCell, version 5

```
IntCell::IntCell (int initialValue)
{ storedValue = new int;
  *storedValue = initialValue; }

IntCell::IntCell (const IntCell & rhs)
{ storedValue = new int;
  *storedValue = *(rhs.storedValue); }

IntCell::~IntCell()
{ delete storedValue; }

void IntCell::operator= (const IntCell & rhs)
{ *storedValue = *(rhs.storedValue); }

int IntCell::read () const
{ return *storedValue; }

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

Note the copy constructor and assignment operator copy what the pointer points to (this is a **deep copy**).

23

Default constructor

- A default constructor is automatically provided if no constructors are provided by the programmer
- It takes no parameters
- For each data member, it
 - ➔ uses defaults for primitive-type data members
 - ➔ uses no-parameter constructor for object-type data members

24

Operator Overloading

- Operators such as =, +, ==, and others can be redefined to work over objects of a class
- The name of the function defining the overloaded operator is `operator` followed by the operator symbol:
`operator+` to overload the + operator, and
`operator=` to overload the = operator
- Just like a regular member function:
 - Prototype goes in the class declaration
 - Function definition goes in implementation file

25

Overload == for IntCel

- Add the prototype to the class decl:

```
bool operator==(const IntCell &rhs);
```
- Add the function definition to the impl file:

```
bool IntCell::operator==(const IntCell &rhs) {  
    return *storedValue == *(rhs.storedValue);  
}
```

- Use `operator==` in another file/function:

```
IntCell object1(5), object2(0), object3;  
if (object2==object3)  
    cout << "object 2 and object3 are equal" << endl;
```

26

Composition

- When one class contains another as a member:

Storage.h: This class declaration uses inlined function definitions, has no .cpp file

```
#include "IntCell.h"  
#include <vector>  
using namespace std;  
  
class Storage  
{  
public:  
    Storage (int size) : cells(size)  
    { }  
  
    int get (int index) const  
    { return cells[index].read(); }  
  
    void put (int index, int value)  
    { cells[index].write(value); }  
  
private:  
    vector<IntCell> cells; // a vector of IntCells  
};
```

27

Composition

- When one class contains another as a member:

Driver.cpp file:

```
#include "Storage.h"  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    Storage memory(10); // calls Storage(int) constructor  
  
    for (int i=0; i<10; i++)  
        memory.put(i,2*i);  
  
    cout << "Cell #4 contents: " << memory.get(3) << endl;  
};
```

```
To compile and run:  
...$ g++ IntCell.cpp Driver.cpp  
...$ ./a.out
```

What is output?

28

enumerated types in C++

- An enumerated data type is a programmer-defined data type.
- It consists of values known as enumerators
- Example:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```
- Day is a data type, it's values are:
 - MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
- Operations that work over enum values:
 - =, ==, !=, <, <=, >, >=, can use as case labels in switch stmt

29

enumerated types in C++

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
Day workday;
workday = WEDNESDAY;
. . .
if (workday==FRIDAY)
. . .
```

- When the enum is declared public inside a class, the typename and values must include the class name and scope resolution operator outside the class

```
class Month {
public:
    enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
    Day whatDayIsIt();
}

Month::Day Month::whatDayIsIt() {
    . . .
    return Month::MONDAY;
}
```

30

random numbers

```
// This program simulates rolling dice.
#include <iostream>
#include <cstdlib> // For rand and srand
#include <ctime> // For the time function
using namespace std;

int main() {
    // Constants
    const int MIN_VALUE = 1; // Minimum die value
    const int MAX_VALUE = 6; // Maximum die value

    // Variables
    int die1; // To hold the value of die #1
    int die2; // To hold the value of die #2

    // Seed the random number generator, so it uses a
    // different sequence of numbers every time program is run
    unsigned seed = time(0); // Get the current system time.
    srand(seed);

    cout << "Rolling the dice...\n";
    die1 = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
    die2 = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
    cout << die1 << endl;
    cout << die2 << endl;
    return 0;
}
```

31