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.

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 functions.

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.
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;
};

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

Using IntCell

```cpp
int main()
{
    IntCell m; // calls IntCell() constructor
    m.write(5);
    cout << "Cell contents: " << m.read() << endl;
    return 0;
};
```

Output:

```
Cell contents: 5
```

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)?
Four changes to IntCell

1. Default parameter
   - IntCell (int initialValue = 0)
   - This constructor has an optional parameter. If not specified, initialValue will be 0.
   ```
   IntCell x;
   IntCell y(5);
   ```

2. Initializer list
   - : storedValue (initialValue)
   - before the constructor body, assigns initialValue to storedValue.
   - sometimes initializer list is required

3. explicit constructor
   - IntCell constructor is labelled “explicit”
   - applies to one-argument constructors only
   - Prevents compiler from doing this conversion:
   ```
   IntCell obj;
   obj = 37;  //should be an error
   ```

4. Constant member function
   - const after param-list declares function will not change any member values: `int read () const`
   - signifies function is an accessor (not a mutator)

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

```
#ifndef _IntCell_H_
#define _IntCell_H_

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

   private:
      int storedValue;
};

#endif
```

Note the "include guards" which prevents 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; }
```

Note the scope resolution operations: `IntCell::` indicates which class the function is a member of.

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);`
    - 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

The Big Three
destructor, copy constructor, operator=

- **Destructor**
  - called when object is destroyed (goes out of scope or deleted)
  - 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=

- **operator=** (aka copy assignment operator)
  - called when = operator is used on existing objects:
    ```
    obj = otherObj;
    ```
  - default operator= applies = to each member
    (aka member-wise assignment)
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.

IntCell, version 4

class IntCell
{
    public:
        explicit IntCell (int initialValue = 0);
        int read () const;
        void write (int x);
    private:
        int *storedValue;
};

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

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

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

What is different from
version 3?

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?

2 4 4 or 2 4 4

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
  making a new copy.
- As a result, all 3 objects point to the same
  location in memory
**IntCell, version 5**

```cpp
class IntCell {
   public:
      explicit 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

**IntCell, version 5**

```cpp
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); };
```

Note: *storedValue = *(rhs.storedValue);
alternatively: write(rhs.read());

**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

**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
Overload == for IntCel

- Add the prototype to the class decl:
  ```cpp
  bool operator==(const IntCell &rhs);
  ```
- Add the function definition to the impl file:
  ```cpp
  bool IntCell::operator==(const IntCell &rhs) {
    return *storedValue == *(rhs.storedValue);
  }
  ```
- Use operator== in another file/function:
  ```cpp
  IntCell object1(5), object2(0), object3;
  if (object2==object3)
    cout << "object 2 and object3 are equal" << endl;
  ```

Exceptions

- An exception is an object that stores information transmitted outside the normal return sequence.
- It is propagated back through calling stack until some function catches it.
- If no calling function catches the exception, the program terminates.
  ```cpp
  int findMax(vector<int> a) {
    int max;
    if (a.size()==0)
      throw "Unable to findMax of empty vector";
    else {
      max = a[0];
      //code to find maximum goes here
    }
    return max;
  }
  ```