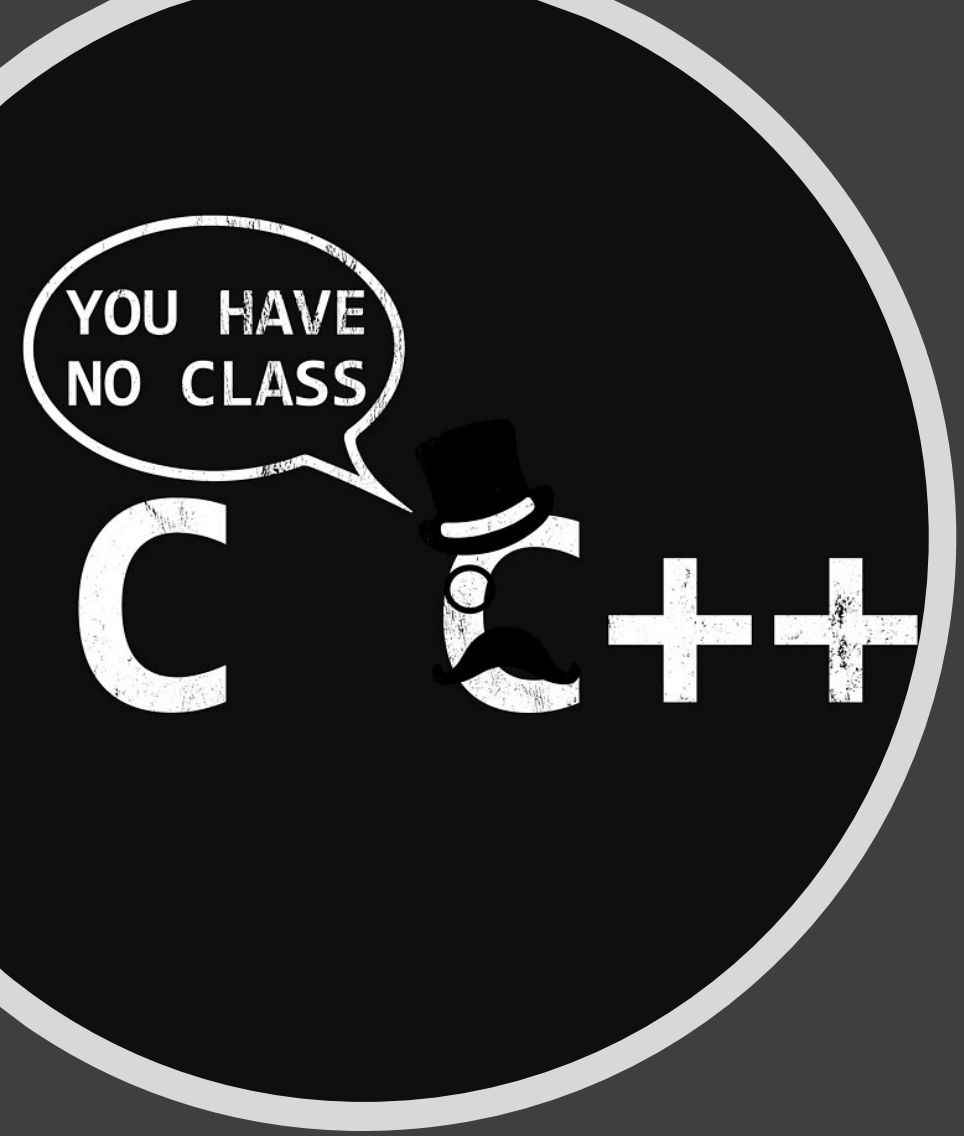

Rick Mercer

C++ Classes

Goals

- Read and understand class definitions
- Implement class member functions using existing class definitions
- Apply some object-oriented design guidelines

Abstraction

- Abstraction is the act of using and/or understanding something without full knowledge of the implementation
- Abstraction allows programmers to concentrate on the essentials
  - how does a `Grid` object move? No need to worry
  - how does `string::substr` work? Don't really need to know
  - How does a vector push_back a value? Maybe we will actually do that later

What is "good" design?

- Design decisions may be based on making
  - a software component more maintainable
  - code that is easier to read and understand
  - software that is easy to use
  - software that can be reused in other applications
- There are usually tradeoffs to consider
- There is rarely a perfect design
- Design is influenced by many things

The C++ Class

- Consider the design of a `BankAccount` type to represent an account at a bank
- Values
  - `name`
  - `balance`
- Operations
  - `deposit`
  - `withdraw`
  - `getBalance`
  - `getName`

Design Decisions with `class BankAccount`

- `BankAccount` could have
  - more operations
  - more values
- `BankAccount` was designed to be simple because it is an introductory example
- An account number wasn't present because
  - it's easier to remember a name like "Smith" rather than a more realistic account number like "217051931"

Class Definitions

- The design a new type can be captured as a class definition

Class Definitions

```
class class-name {
public:
  class-name() ;                    // Default constructor
  class-name(parameter-list);       // Constructor with parameters
  function-heading;                 // Member functions that modify state
  function-heading;
  function-heading const;           // Members function that don't modify
  function-heading const;
private:
  object-declaration;               // Data members -- the state
  object-declaration;               // that can also be initialized here
};
```

Some objects need 2 or more arguments in the constructor

- `int`, `double`, and `string` objects are initialized with only one argument and optional use `( )`

```
int n = 0;                  int n(0);
double x = 0.001;           double x(0.001);
string s = "Kim";           string s("kim");
```

- `BankAccount` objects require two arguments, initialization with = is not an option
- The special function is named `BankAccount`

```
BankAccount anAccount("Kim", 100.00);
```

Class Definition (comments removed)

```
#include <string>

class BankAccount {
public:
  BankAccount();
  BankAccount(const std::string &initName,
              double initBalance);

  void deposit(double depositAmount);
  void withdraw(double withdrawalAmount);

  double getBalance() const;
  std::string getName() const;

private:
  std::string name;
  double balance;
};
```

Class Definitions

- The following things can be determined from a class definition
  - The class name
  - The name of all member functions
  - The return type of any function (or if it is void)
  - The number and type of arguments required in any member function call
  - The action of each member function *if it has comments, that is*

Class Definitions

- Class definitions
  - represent the interface, which is the collection of available messages
  - describe the member function headings to enable syntactically correct messages
  - describes the data members
    - the values each object will remember

Objects are about Operations and State

- The *function-headings* after `public:` represent the messages that may be sent to any object
- The *data-members* after `private:` store the state of any object
  - every instance of a class has it own separate state

Construct an object, send Messages

```
// This code would compile, but not build until
// the methods are implemented in BankAccout.cpp
#include <iostream>
#include "BankAccount.h"  // for class BankAccount

int main() {
  BankAccount anAcct ("Alex", 50.00 ); // Construct

  std::anAcct.withdraw(20.00);  // Modify
  anAcct.deposit(40.00);        // Modify
  std:: cout << anAcct.getName() << endl;    // Access
  std:: cout << anAcct.getBalance() << endl; // Access

  return 0;
}
```

Output

```
Alex
70
```

Implementing Class Member Functions

- Class member function implementation are similar to their non-member counterparts
- All class member functions must be qualified with the class name and `::` scope resolution operator
  - Important! This gives the member functions access to the private data members.
- Constructors have the same name as the class and no return type
- Modifying member functions can not have `const`
- Accessing member functions should have `const`

Why have .h files

- The practice of studying a class through its interface represents a principle in software engineering
  - This allows us to separate the interface from the implementation, that are the details in the functions
- In C++, interfaces are in header (`.h`) files
- Member function implementations are separated from class definitions in a `.cpp` file
- Using .h file speeds up compilation (large programs)
- It is easier to understand a type by looking at the interface rather than the implementation

Implementing Constructors

- The following constructor with parameters is called whenever objects are constructed like this

```
BankAccount anAccount("Mason", 2500.00);
```

```
// This code is in the file BankAccount.cpp
#include "BankAccount.h" // Get the definition

BankAccount::BankAccount(const std::string &initName,
                         double initBalance) {
  name = initName;
  balance = initBalance;
}
```

- The parameters are used to initialized the private instance variables `name` and `balance`

Constructors

- Constructors
- They differ from the other member functions
  1. they have no return type
  2. they have the same name as the class

Default Constructors

- Classes can have more than one constructor
- Default constructors assign default values to the private instance variables
- A default constructor is a constructor with no parameters and no return type

```
BankAccount::BankAccount() {
  name = "?";
  balance = 0.0;
}
```

Why Default Constructors?

- Default constructors are required to have collections of objects; needed later with `vector` objects
- Default constructors guarantee initialization to a specific state so programmers always know what to expect (more vivid examples are yet to come)
- Default constructors define the default values used when another default constructor is called
  - For example, the default state for string is the empty string " "

Implementing Modifiers

- Member functions are implemented like free functions and qualified with *class-name*`::`
  - The scope resolution operator `::` gives the modifier access to the private instance variables that need to be modified
  - In modifying member functions, the private data member `balance` is changed so do ***not*** use `const`

```
void BankAccount::deposit(double depositAmount) {
  balance = balance + depositAmount;
}
void BankAccount::withdraw(double amount) {
  balance = balance - amount;
}
```

Implementing Accessor Functions:

- Accessor functions must also be qualified with *class-name* :: to gives access to the state being used to return info *remember to write* `const`

```
double BankAccount::getBalance() const {
  return balance;
}

std::string BankAccount::getName() const {
  return name;
}
```

- The state is now available with these messages

```
cout << anAcct.getBalance() << endl;
cout << anAcct.getName() << endl;
```

Construct Objects

- Create 3 default `BankAccount` objects that would have an initial balance of 0.0 and a name of "?"

```
BankAccount a, b, c;
```

- Initialize `BankAccount` objects with `( )` or `{ }`

```
BankAccount anAcct {"Kim", 123.45};
BankAccount anotherAcct ("Chris", 200.00};
```

Constructing Objects

- General form for object construction

  type identifier(s);

  -or-

  type identifier(*initial-state*);

- When passing one or more arguments to a constructor, enclose the arguments in ( ) or { }

```
string name("First I. L. Last");
string name2{"Last, First"};
BankAccount anAcct("Alex", 50.00);
BankAccount anAcct{ "Alex", 50.00};
```

Constructing Objects

- C++ default constructor calls keep the same format at creating objects from C++ primitives:
- When calling the default constructor to create one or more objects, do not use the ()

```
string s1, s2, s3;
BankAccount a, b, c;
```

- These are incorrect

```
string s1(), s2(), s3();   // Wrong
BankAccount a(), b(), c(); // Wrong
```

Object-Oriented Design Heuristics

- Classes must be designed
  - function names, needed parameters and return types
- There are some heuristics (guidelines) to help us make design decisions

  Design Heuristic

  All data should be hidden within its class

- Ramifications
  - Good: Can't mess up the state (compiler complains)
  - Good: Have to create interface of member functions
  - Bad: Extra coding, but worth it

Cohesion Within a Class

- A class definition provides the public interface
  - The methods and state should be closely related
- The related data objects necessary to carry out a message should be in the class

Design Heuristic

Keep related data and behavior in one place

- Ramifications
  - Good: Provides intuitive collection of operations
  - Good: Reduces the number of arguments in messages
  - Bad: None that I can think of

Cohesion

- Synonyms for cohesion:
  - hanging together
  - unity, adherence, solidarity
- Cohesion means
  - data objects are related to the operations
  - operations are related to the data objects
  - data and operations are part of the same class definition

Cohesion

- For example, cohesion means
  - `BankAccount` does not have operations like `dealCardDeck` or `ambientTemperature` or instance variables `velocity` or `meters`
- `BankAccount` member functions have access to `balance`, something which often needs to be referenced
  - The private instance variable `balance` is not maintained separately or passed as an argument

`const` or not?

- Note that `const` follows the accessing functions but not the modifying functions
- This makes our object 'safer' by avoiding accidental modification
- Using `const` is necessary to allow objects be passed by `const` reference to another function

`const` Messages are Okay, Non `const` are Errors

```
void display(const BankAccount & b) {
  // OKAY to send name and balance messages since they
  // were both declared with const member functions
  cout << "{ BankAccount: " << b.getName()
       << ", $" << b.getBalance() << " }" << endl;

  // Modifying message to non-const member function
  // was not tagged as const. It should be an ERROR
  b.withdraw(234.56); // <- Error
}
```

A C++ Specific Guideline

- This leads to another guideline that is particular to C++

  Design Heuristic

  Always declare accessor member functions as const, Never declare modifiers as const

- This guideline is easy to forget
  - Unless you thoroughly test, you may not get a compiletime error
  - You will not be told something is wrong until you try to pass an instance of your new class by `const` reference

Naming Conventions

- Rules #1, 2, and 3:
  1: Always use meaningful names
  2: Always use meaningful names
  3: Always use meaningful names
- Rule #4

  Constructors: Name of the class
  Modifiers: Verbs `borrowBook withdraw`
  Accessors: Nouns with get `getLength getName`

`public:` or `private:`

- When designing a class, do this *at least for now*
  - place operations under `public:`
  - place object that store state under `private:`
- Public messages can be sent from the block in which the object is declared
- Private state can not be messed up like this

```
BankAccount myAcct("Me", 10.00);
myAcct. balance = myAcct. balance + 999999.99;
```