Intro, Mechanics, Classes & Objects

Tonight's Plan

- Reading : 393-410
- Abstraction and programming styles [4-6, 394-395]
- Syllabus and class mechanics
- Programming mechanics
- What are classes? [395-400]
- Implementing member functions [400-405]
- Creating and using objects [384-387]
- Homework 1 and 2

Why C++?

- Why are there different computer languages?
  - Does the computer understand programs written in C++ better than those written in Visual Basic, Pascal or Java?
  - No. The computer doesn't "understand" C++ or Pascal. The computer only understands programs written in its native language: machine language.
- So? Why don't we use machine language?
Abstraction I

We avoid machine language programs because we have difficulty reading and writing them. For humans, machine language programs are just too complex.

According to a famous study by George Miller, [1956] our brain can only manage to deal with about seven [plus or minus two] units of memory [chunks] at a time.

We handle more info using abstractions

Abstraction II

What is an abstraction?

- The ability to combine small chunks of information into larger chunks.

Example: 1949661328321173

- What does this number mean?
- Could you learn this number?

Now? 1 (949) 661-3283 x 21173
Generation 2 - Translators I

- First abstraction groups bits together
  - Is this a big improvement?

  11AA:0000 0E 1F BA 0E 00 00 00 CD -21 B8 01 4C CD 21 54 68
  11AA:0010 69 73 20 70 72 6F 67 72 -61 6D 20 63 61 6E 6E 6F
  11AA:0020 20 62 65 20 72 75 6E -20 69 6E 20 44 4F 53 20
  11AA:0030 0D 00 00 00 00 00 00 00 -24 00 00 00 00 00 00 00
  11AA:0040 0D 00 00 00 00 00 00 00 -24 00 00 00 00 00 00 00
  11AA:0050 0D 00 00 00 00 00 00 00 -24 00 00 00 00 00 00 00
  11AA:0060 0D 00 00 00 00 00 00 00 -24 00 00 00 00 00 00 00

Generation 2 - Translators II

- Second abstraction replaces machine instructions with "mnemonic" labels
  - A "translator" program then converts those labels to machine language

  11AA:0000 0E            PUSH    CS
  11AA:0001 1F            POP     DS
  11AA:0002 BA0E00        MOV     DX,000E
  11AA:0005 B409          MOV     AX,09
  11AA:0007 CD21          INT    21
  11AA:0009 80014C        MOV     AX,4C01
  11AA:000C CD21          INT    21

Generation 2 - Translators III

- This mnemonic style of machine language is called assembly language
  - Easier for humans to understand
    - A single chunk hides more details

- Third abstraction: combining several mnemonic instructions into one mnemonic instruction
  - These are called macros
  - Translators that use them are macro assemblers
Generation 3 - Idea Translation

- Second-generation programmers had to learn to "think like a computer"
  - Required two different translations
  - **Step 1:** Solve the problem logically
    - Define the data needed and the algorithms to process that data (How to calculate sales tax, for instance)
  - **Step 2:** Translate to computer instructions
- Third-generation translators targeted Step 2
  - Included COBOL, FORTRAN, C, and Pascal

Why C++ (Again)?

- So? Why don't we all program in C? Why do we have a multitude of 3GL languages?
  - Different languages for different domains
    - COBOL for business, FORTRAN for engineering
  - Main reason? Programs have gotten larger
    - Larger programs have too many "chunks"
    - Humans need additional abstraction levels to create, understand, and maintain them
    - These abstractions must target Step 1

Six Principles of Abstraction

- Principles used to hide complexity
  - Divide and Conquer [decomposition]
  - Encapsulation [cohesion]
  - Interface [coupling]
  - Information hiding [modularity]
  - Generality [refinement]
  - Extensibility [reuse]
Principles II

- **Divide and Conquer**
  - Break big problems into smaller "parts"
  - This is called decomposing the problem
- **Encapsulation**
  - "Package" similar or related parts together
  - Functions that do this are cohesive
- **Interface**
  - Simplify communication between unrelated parts
  - Passing parameters, reduces such coupling

Principles III

- **Information Hiding**
  - Group related parts into sub-systems
  - Limit access parts inside of a subsystem
- **Generality**
  - Group common operations together
  - This is called refinement or factoring out
- **Extensibility**
  - Don't build each part from scratch
  - Reuse existing parts as a base for new parts

Why C++ (Yet Once Again)?

- **C++ is based upon C**
  - Same types, keywords, control mechanisms
  - C++ adds new features to C
- **Support for different abstractions**
  - Procedural and modular programming [C]
  - Object-based programming
  - Object-oriented programming
  - Generic programming
C++ Programming Styles I

- **Procedural style**
  - "Decide which procedures you want; use the best algorithms you can find."
  - Focus is on processing, finding algorithm
  - Language support:
    - Local variables
    - Passing values to functions, type checking
    - Returning values from functions
  - Principle: decomposition, encapsulation, interface

C++ Programming Styles II

- **Modular Style**
  - "Decide which modules you want; partition program so data is hidden within modules"
  - Focus shifts from algorithm to data
  - Style used for almost all larger programs
    - Example: Word processor, spreadsheet
    - Built around design of document
  - Language support: separate compilation
  - Principle: data hiding, interface

C++ Programming Styles III

- **Object-based Style [User-defined types]**
  - "Decide which types you want; provide a full set of operations for each type"
  - Focus shifts to combination of data and action
  - Language support:
    - Classes combining data and functions
    - Overloaded operators so new types can act like the built-in types
  - Principles: encapsulation, interface, data-hiding
C++ Programming Styles IV

- **Object-oriented Style**
  - "Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance"
  - Focus shifts from object relationships to relationships between classes
    - Language support:
      - Inheritance, virtual functions, abstract classes
    - Principle: extensibility, interface, generality

C++ Programming Styles V

- **Generic Style**
  - "Decide which algorithms you want; parameterize them so they work for a variety of suitable types"
  - Focus switches back to algorithms and reuse
  - Language support:
    - Template functions
    - Template classes
  - Principle: generality

Syllabus

- Class Info and Mechanics
Syllabus I

- **Class Web Site**
  - csjava.occ.cccd.edu/~gilberts/

- **Meeting location & time**
  - Monday, 5:00-10:25, Business Education 106
  - Weekly: 3.5 hr Lecture, 1.5 hr discussion
  - General plan for lectures/breaks
    - Session 1: 5:00-6:30 pm (1.5 hours)
    - Session 2: 6:45-8:00 pm (1.25 hours)
    - Session 3: 8:15-9:15 pm (1 hour)
    - Session 4: 9:30-10:25 pm (55 minutes)

Syllabus II

- **Instructor**
  - Stephen Gilbert
  - Preferred communication, email:
    - sgilbert@occ.cccd.edu
    - StephenDGilbert@netscape.net
  - Phone: (714) 432-0202 ext 21173
  - In person office hours:
    - Clark Computing Center F
    - Monday, Wednesday: 4:00-5:00 pm
    - Tuesday, Thursday: 1:30-2:00, 4:30-5:00

Syllabus III

- **Course description**
  - Is this course for you?
  - Goals & Outcomes

- **Course requirements**
  - Reading & workload
  - Exams
  - Programming assignments
  - Grades
Syllabus IV

- Class policies
  - Late work
  - Absences
  - Academic honesty
  - Disruptive behavior
  - Disabilities
  - Changes to the syllabus

Syllabus V

- Course resources
  - Textbook
    » Stephen Prata's, C++ Primer Plus, 4th Ed
    » Optional books
  - Software
    » DevC++/MinGW (GCC)
    » Quincy/GNU GCC Compiler
    » Visual C++ 7.1 (Visual Studio .NET)

Class Schedule

Student Accounts

- Need an account to submit your assignments
  - ID: CS250-XXX
    » XXX = three-digit number between 100 and 199
  - Password
    » Case sensitive, 1 capital, one number
    » Watch out for el and one, zero and oh
  - Nickname
    » Chosen at random from dictionary
    » Used when posting your grades
C++ Mechanics

Edit, Compile, & Link

Getting Started

- Most CS 250 projects involve multiple files
  - You can use an IDE or command-line compiler
- You must use an ANSI/ISO compiler
  - GNU C++ 3x (DevC++ 5x, MingW 3x, Quincy 2002)
  - Cannot use Visual C++ 6, but VS.NET (7.1) is OK
  - Borland C++ (on Web) works for 90%
  - All of these are available in the Computing Center
  - See Web links for installing and configuring
- See links on Web site for Mac/Linux links

Create your Source Code

- Use IDE or editor to create source code files
  - Class declarations will go in a header file
    - One header file for each class you create
  - Class definition (implementation) for functions
    - Once .cpp file for each class
  - Client file will contain your main() function
- Examples: stock1.h, stock1.cpp, usestock1.cpp
  - Page 416-418 in your text, Week 2 Code folder
**GCC Compile & Link**

- **Using MinGW or GCC command line tools**
  - Download and install instructions on Web page
  - Compile & link all files (on one line)
    - `g++ -ansi -pedantic -Wall -o usestok1.exe stock1.cpp usestok1.cpp`
  - Compile individual files [with separate linking]
    - `g++ -ansi -pedantic -Wall -c stock1.cpp`
    - `g++ -ansi -pedantic -Wall -c usestok1.cpp`
    - `g++ -o usestok1.exe usestok1.o`

**DevC++ Compile & Link**

- **Download & Configure Instructions on Web**
  - 1. Create a new Project for each assignment
    - Choose Empty Project from the New Project dialog
    - Create folder to store your project, save project file
  - 2. Create the source code files
    - Right-click project and choose New File
    - Save file with desired name
  - 3. Compile button compiles and builds
    - Check progress using Compile Log tab

**Quincy 2002 Compile & Link**

- **Using Quincy 2002 with MinGW compiler**
  - 1. Create a Project
    - Choose File | New, Choose Project from dialog
    - Type name of EXE in Target text area
    - Select Console Application and click OK
  - 2. Create source code files and add to project
    - Can add all files to project at once
  - 3. Click Project-Build or click Build/Rebuild button
VC7 Compile & Link I

Using VC++ 7x (Visual Studio .NET)
- Available for checkout and install at home
- Configuration instructions on Web site as well
- 1. Create a Project
  » Close all then choose File | New | Project from menu
  » Choose Visual C++ Projects in the New dialog
  » Choose Win32 Console App from Templates
  » Enter name of executable as Project Name
    » No extension when you enter name
    » Change location if you desire

VC7 Compile & Link II

Using VC++ 7x (Visual Studio .NET)
- 2. When AppWizard appears:
  » Choose Application Settings from list on left
  » Check Empty Project and click OK
- 3. Set project settings for CS 250
  » Locate project in Solution Explorer
  » Rightclick and choose Properties
  » Select C++ Language, and turn OFF extensions
  » Turn on correct "for" scope, RTTI and exceptions

VC7 Compile & Link III

Using VC++ 7x (Visual Studio .NET)
- 4. Add source files to project
  » In Solution Explorer, rightclick Source folder
  » Choose Add New Item or Add Existing Item
  » You can add header files
- 5. Build project from the Build menu
  » Choose Build Solution
  » To run interactively, choose Run without Debugging
Running Your Programs

- Most CS 250 homework will use redirection
- Running from the command-line
  - Locate the Command Prompt or MS-DOS Prompt
  - Use CD to navigate to location of executable
- To use hw02.txt as input to hw02.exe
  - `C:\docs> hw02 < hw02.txt`
- To send the output of hw02 to output.txt
  - `C:\docs> hw02 > hw02-output.txt`

Homework 1

- Assignment 1: Using your compiler
  - Due before next Monday, 2/8/2004
  - Submit before 2 am, Monday AM
  - Use your editor to create three text files
    - `mytime1.h` [page 450], `mytime1.cpp` [page 451], `usetime1.cpp` [page 452] (check errata first!)
    - Add your name & ID to the top of each listing
  - Change values of B & C in usetime1.cpp
  - Submit usetime1.cpp

Object-Based Programming

Creating User-defined Types
Object-Based Programming

- **Purpose:** to create user-defined types
  - Allows us to extend the language
  - New types will be "first-class" types
    - Can do anything a built-in type can do (actually more)
- **What do we need to do?**
  - Define new types, called classes
  - Write code for new class behavior
    - Called member functions or methods
  - Create and use instances of types, called objects

Categories of Data Types

- **C/C++** have three categories of data types
  - Built-in or primitive types
    - Int, char, double, etc, just like C
    - One new primitive: bool, values true/false
  - Built-in derived types with language support
    - Arrays and C-style strings
  - User-defined types
    - User-defined just means "not built-in"

C and C++ User-Defined Types

- **In C,** there are four user-defined types
  - struct, union, typedef, enum
  - C++ adds the class type to these four
- **In C,** only primitives are "first-class" types
  - Cannot use assignment for arrays and strings
  - Cannot compare structs for equality
  - Can assign non-enum values to enum variables
**Advantage of User-defined Types**

- In C++, classes and structs are first-class types
  - Can be made to act just like primitive types
  - This is the goal of object-based programming
- Actually more powerful than primitive types
  - Objects have complete control over initialization
  - Primitive can only be manipulated via operators
  - Objects are used in two ways: operators & methods
  - Unlike primitives, object behavior is extensible

**Classes and Objects**

- C++ types like string are called classes
  - Defining a class is like creating a struct, but
    - You declare both data members and functions
    - You specify the "level" of access
    - You define the function bodies separately
- The class acts like a blueprint
  - Allows you to create many objects
  - Describes what the objects can do
  - Similar to creating structure variables

**Creating a Class**

- Class specification is like a struct specification
  - struct specifier is used to create structure variables
- Class specification has two parts
  - Class declaration [sometimes called the interface]
    - Data declarations [just like a struct]
      - Called data members
  - Class function prototypes
    - Called member functions, or methods
  - Method definitions, usually placed in another file
Class Declaration I

- Let's create a simple class that represents Ducks
  - Every Duck has a name (this is its data or state)
  - You can change its name, retrieve its name, or ask the Duck to introduce itself.
- Create a specifier, use class, not struct
  - Declare both data and functions [methods]

```cpp
class Duck {
  // Declare data members
  // Declare function prototypes
};
```

Class Declaration II

- Divide specification into two sections:
  - The private: section. Declare data here
    - Only accessible from methods [member functions]
  - The public: section. Prototype functions here
    - The public section is called the class interface

```cpp
class Duck {
  private:
    // Declare data members
  public:
    // Declare functions
};
```

Class Declaration III

- Define the data members
  - We'll use one string member for the Duck's name
    - Need to include Standard C++ <string> header
    - Use fully qualified name for data member

```cpp
#include <string>
class Duck {
  private:
    std::string name;
  public:
    // Declare functions
};
```
Class Declaration IV

- Define the member functions
  - "Get" and "Set" functions to retrieve/change data
    » Cannot retrieve it directly, as with a structure
  - A "speak" function to introduce the Duck.

```cpp
#include <string>

class Duck {
private:
  std::string name;
public:
  std::string get_name();
  void           set_name(std::string n);
  void           speak();
};
```

Class Declaration V

- Declaration is usually placed in a header file
  - Often called the class interface file
  - Method defs usually placed in implementation file
  - User includes header when creating objects [client]
  - Program links code where methods defined

```
// Duck.h - method definition for Duck class
#include <string>
#include <iostream>
using namespace std;

// Define each method in Duck class here
```

Class Definition I

- Class definition contains method code
  - Usually in a .cpp file of same name
  - Starts by including declaration file
    » Use quotes when including user-defined headers
    » Include necessary standard library headers as well
    » OK to include using directive in implementation file

```
// Duck.cpp - method definitions for Duck class
#include "Duck.h"
#include <string>
#include <iostream>
using namespace std;

// Define each method in Duck class here
```
**Class Definition II**

Defining a method is similar to defining a function
- 1) Make copy of original prototype
- 2) Add argument names if necessary
- 3) Replace the ending semicolon with braces
- 4) Add your code inside the braces

Scope of method is specific class
- Method name must be prefixed by `class-name::`

Methods can access entire class
- Both private data and public methods

**Class Definition III**

Example: the `set_name()` method
- 1. Copy the prototype from the interface file
   - `void set_name(string n);`
- 2. Add the scope-resolution operator
   - `void Duck::set_name(string n);`
- 3. Replace semicolon with braces
   - `void Duck::set_name(string n) {}`
- 4. Add code
   - `void Duck::set_name(string n)`
   
**Inline Member Functions**

You can create inline member functions as well
- For very simple methods that must execute quickly

Two ways to create inline methods:
- 1) Define body inside class declaration
   - Called "automatic" inlining
- 2) Declare in class, define outside
   - Use the inline keyword when defining
   - Definition must appear in header file, not with other member functions in the implementation file.
Using Objects I

- Let's create a program that uses the Duck class
  - Such a program is often called a "client" program
- We need to do four things:
  - 1) Include the header [declaration] file Duck.h
    - Allows our program to create Ducks without definition
  - 2) Create Duck objects in our program
    - This is called instantiation
  - 3) Send messages to our Duck objects
  - 4) Link our client code to the Duck definition code

Using Objects II

- Step 1: Setting up the program environment
  ```cpp
  // UseDuck.cpp - use the Duck class
  #include "Duck.h"
  #include <iostream>
  using namespace std;
  int main()
  {
    // Create and use Duck objects here
    return 0;
  }
  ```

Using Objects III

- Step 2: Create some Duck objects
  - Remember that objects are just like variables
  ```cpp
  int main()
  {
    // Create and use Duck objects here
    Duck don, daf;
    return 0;
  }
  ```
Using Objects IV

- Step 3: Send “messages” to objects
  - Can call any method in public interface
  - Syntax similar to accessing structure data members
    - `object + "dot" + function call`

```cpp
int main()
{
    Duck don, daf;
    don.set_name("Donald Duck");
    don.speak();
    return 0;
}
```

---

Homework

- Assignment 2: Your first class
  - Due Sunday, February 8. [Grade Mon morn.]
  - Download `Fraction01.h`, `hmwk01.cpp`, and `hmwk01.txt`
  - Create `Fraction01.cpp`
  - Implement functions declared in `Fraction` class
  - Compile & link with `hmwk01.cpp`
  - Test with `hmwk01.txt`
  - Submit only `Fraction01.cpp`