Testing and Debugging
I/O Streams
Intro to OOP Concepts

CS 16: Solving Problems with Computers I
Lecture #10

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- **Homework #9 due today**
- **Lab #5 is due on Friday at Noon**
- Your grades are NOT ON GAUCHOSPACE anymore. Instead go to:

http://cs.ucsb.edu/~zmatni/cs16/CS16Grades_Fa2016.htm

Lecture Outline

- Testing & debugging techniques
- I/O streams
- An introduction to
  Object Oriented Programming (OOP) concepts

Testing and Debugging Functions

- Each function should be tested as a separate unit
- Testing individual functions facilitates finding mistakes
- “Driver Programs” allow testing of individual functions
- Once a function is tested, it can be used in the driver program to test other functions

Example of a Driver Test Program

```
int main( )
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
             << wholesale_cost << endl;
        cout << "Days until sold is now "
             << shelf_time << endl;

        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}
```

Stubs

- When a function being tested calls other functions that are not yet tested, use a **stub**
- A stub is a *simplified version of a function*
- Stubs are usually **provide values for testing** rather than perform the intended calculation
  - i.e. they're fake functions
- Stubs should be so simple that you have confidence they will perform correctly

Stub Example

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}

//Uses iostream:
void give_output(double cost, int turnover, double price)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Wholesale cost = $" << cost << endl
         << "Expected time until sold = "
         << turnover << " days" << endl
         << "Retail price= $" << price << endl;
}

//This is only a stub:
double price(double cost, int turnover)
{
    return 9.99; //Not correct, but good enough for some testing.
}
```

fully tested function

function being tested

stub

Fundamental Rule for Testing Functions

Test **every function** in a program in which

every *other* function in that program

has already been fully tested and debugged

Debugging Your Code

- Keep an open mind
  - Don’t assume the bug is in a particular location

- **Don’t randomly change code** without understanding what you are doing until the program works
  - This strategy may work for the first few small programs you write but it is doomed to failure for any programs of moderate complexity

- Show the program to someone else

General Debugging Techniques

- Check for common errors, for example:
  - Local vs. Reference Parameters
  - = instead of ==
  - Did you use && when you meant ||?
  - These are typically errors that might not get flagged by a compiler

- Localize the error
  - Narrow down bugs by using **`cout`** statements to reveal internal (hidden) values of variables
  - Once you reveal the bug and fix it, remove the **cout** statements

Example: Debug this Program

```
 1    #include <iostream>
 2    using namespace std;
 3
 4    int main()
 5    {
 6        double fahrenheit;
 7        double celsius;
 8
 9        cout << "Enter temperature in Fahrenheit." << endl;
10        cin >> fahrenheit;
11        celsius = (5 / 9) * (fahrenheit - 32);
12        cout << "Temperature in Celsius is " << celsius << endl;
13
14        return 0;
15    }
```

Sample Dialogue

```
Enter temperature in Fahrenheit.
100
Temperature in Celsius is 0
```

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6       double fahrenheit;
7       double celsius;
8
9       cout << "Enter temperature in Fahrenhei
10      cin >> fahrenheit;
11
12      // Comment out original line of code but leave it
13      // in the program for our reference
14      // celsius = (5 / 9) * (fahrenheit - 32);
15
16      // Add cout statements to verify (5 / 9) and (fahrenheit - 32)
17      // are computed correctly
18      double conversionFactor = 5 / 9;
19      double tempFahrenheit = (fahrenheit - 32);
20
21      cout << "fahrenheit - 32 = " << tempFahrenheit << endl;
22      cout << "conversionFactor = " << conversionFactor << endl;
23      celsius = conversionFactor * tempFahrenheit;
24      cout << "Temperature in Celsius is " << celsius << endl;
25
26      return 0;
27  }
```

Sample Dialogue

```
Enter temperature in Fahrenheit.
100
fahrenheit - 32 = 68
conversionFactor = 0
Temperature in Celsius is 0
```

code that is commented out

debugging with cout statements

Other Debugging Techniques

- Use a debugger tool
  - Typically part of an IDE (integrated development environment)
  - Allows you to stop and step through a program line-by-line while inspecting variables

- Use the assert macro
  - Can be used to test pre or post conditions

```
#include <cassert>
assert(boolean expression)
```

  - If the boolean is false then the program will abort
    - **Not** a good idea to keep in the program once you're done

Assert Example

- Denominator should not be zero in Newton's Method

```
// Approximates the square root of n using Newton's
// Iteration.
// Precondition: n is positive, num_iterations is positive
// Postcondition: returns the square root of n
double newton_sqroot(double n, int num_iterations)
{
    double answer = 1;
    int i = 0;

    assert((n > 0) && (num_iterations> 0));
    while (i <num_iterations)
    {
        answer = 0.5 * (answer + n / answer);
        i++;
    }
    return answer;
}
```

I/O Streams

- **I/O** = program Input and Output

- Input can be delivered to your program via a *stream object*
- This is when input can be from:
  - The keyboard
  - A file
- Output is delivered to the *output device* via a stream object
- Output devices can be:
  - The screen
  - A file

Objects

- Objects are special variables that have their own special-purpose functions
  - Example: string length can be gotten with **`stringname.size()`**
  - These are called *member functions*

Streams and Basic File I/O

- Files for I/O are the same type of files
  used to store programs

- A stream is a *flow of data*

- Input stream: Data flows *into* the program

- Output stream: Data flows *out of* the program

cin And cout Streams

- **cin**
  - Input stream connected to the keyboard
- **cout**
  - Output stream connected to the screen

- cin and cout are defined in the iostream library
  - Use include directive: #include <iostream>

- You can also use streams with *files*

Why Use Files?

- Files allow you to store data permanently!
- Data output to a file lasts after the program ends
  - You can usually view them without the need of a C++ program
- An input file can be used over and over
  - No typing of data again and again for testing
- Create or read files at your convenience
- Files allow you to deal with larger data sets

File I/O

- Reading from a file
  - Taking input from a file
  - Done from beginning to the end (not always)
    - No backing up to read something again (but OK to start over)
    - Similar to how it's done from the keyboard
- Writing to a file
  - Sending output to a file
  - Done from beginning to end (not always)
    - No backing up to write something again (but OK to start over)
    - Similar to how it's done to the screen

Stream Variables for File I/O

Like other variables, a stream variable…

- Must be **declared** before it can be used
- Must be **initialized** before it contains valid data
  - Initializing a stream means *connecting it to a file*
  - The value of the stream variable is really the file it is connected to
- Can have its value changed
  - Changing a stream value means disconnecting from one file and then connecting to another

Streams and Assignment

- A stream is a special kind of variable called an object
  - Objects can use special functions to complete tasks

- Streams use special functions instead of the assignment operator to change values

- Example:

```
streamObjectX.open("addressBook.txt");
streamObjectX.close();
```

Declaring An Input-file Stream Variable

- Input-file streams are of type **ifstream**

- Type **ifstream** is defined in the **fstream** library
- You must use the include and using directives

```
#include <fstream>
using namespace std;
```

- Declare an input-file stream variable with:

```
ifstream in_stream;
```

Declaring An Output-file Stream Variable

- Ouput-file streams of are type **ofstream**

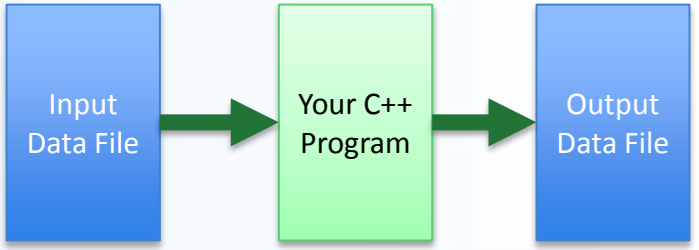
- Type **ofstream** is defined in the **fstream** library
- Again, you must use the include and using directives

```
#include <fstream>
using namespace std;
```

- Declare an input-file stream variable using

```
ofstream out_stream;
```

Connecting To A File



- Once a stream variable is declared,
  you connect it to a file
  - Connecting a stream to a file means “opening” the file
  - Use the *open* function of the stream object

```
in_stream.open("infile.dat");
```

Period

Double quotes

File name on the disk

Using The Input Stream

- Once connected to a file, get input from the file using the extraction operator (**>>**)
  - Just like how you do that with **cin**

Example:

```
ifstream in_stream;
int one_number, another_number;
in_stream >> one_number >> another_number;
```

Using The Output Stream

- An output-stream works similarly using the insertion operator (**<<**)
  - Just like how you do that with **cout**

Example:

```
ofstream out_stream;
out_stream.open("outfile.dat");

out_stream << "one number = "
           << one_number
           << ", another number = "
           << another_number;
```

External File Names

An External File Name…

- Is the name of a file that the operating system uses
  - *infile.dat* and *outfile.dat* used in the previous examples
- Is the "real", on-the-disk, name for a file
- Needs to match the naming conventions on your system
  - Don't call an input **text** file *XYZ.jpg*, for example…
- Usually only used in the stream's open statement
  - ***Example:*** `in_stream.open("infile.dat");`
- Once open, it is referred to with the name of the stream connected to it
  - ***Example:*** `in_stream >> VariableX;`

Closing a File

- After using a file, it should be closed using the .close() function
  - This *disconnects* the stream from the file
  - Close files to reduce the chance of a file being corrupted if the program terminates abnormally

- ***Example:*** `in_stream.close();`

- It is important to close an output file if your program later needs to read input from the output file

- The system will automatically close files if you forget

as long as your program ends normally!

Objects

- An object is a variable that has functions and data associated with it
  - **in_stream** and **out_stream** each have a function named ***open*** associated with them
  - **in_stream** and **out_stream** use *different versions* of a function named ***open***
    - One version of open is for input files
    - A different version of open is for output files

Member Functions

- A *member function* is a function associated with an object
  - The ***open*** function is a member function of **in_stream** in the previous examples
  - Likewise, a *different* ***open*** function is a member function of **out_stream** in the previous examples
  - Same for the ***close*** function

- For a list of member functions for I/O stream classes, see:

http://www.cplusplus.com/reference/fstream/ifstream/

http://www.cplusplus.com/reference/fstream/ofstream/

Objects and Member Function Names

- Objects of different types
  have different member functions
  - Some of these member functions might have the same name

- Different objects of the same type
  have the same member functions

Classes vs. Objects

- A type whose variables are objects, is a class
  - **ifstream** is the type of the **in_stream** variable (the *object*)
  - ifstream is a *class*
  - The class of an object determines its member functions
  - Example:

    ifstream in_stream1, in_stream2;

    - in_stream1.open and in_stream2.open
      are the *same function* (because they are the same class)
      but might have *different arguments*

Class Member Functions

- Member functions of an object
  are the member functions of its class

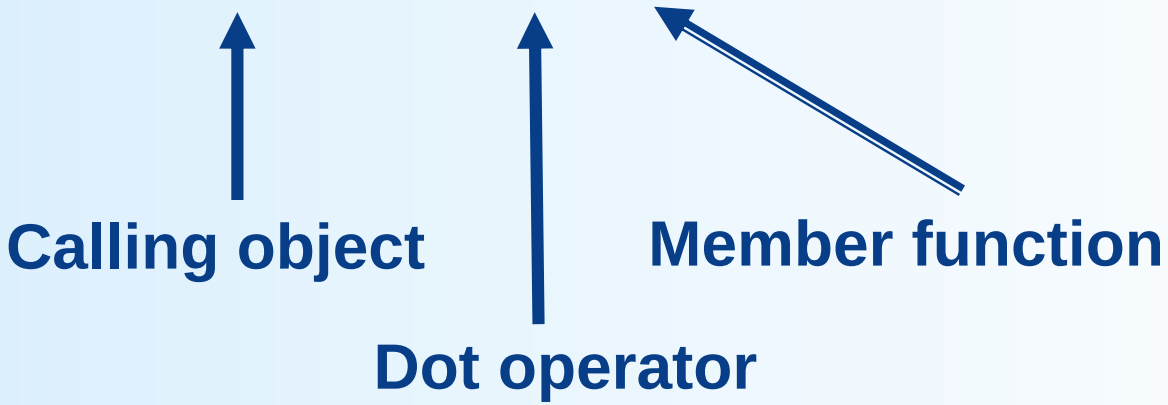
- The class determines the member functions that an object can use
  - The class **ifstream** has an ***open*** function
  - Every variable (object) declared of type **ifstream** also has that ***open*** function

Calling a Member Function

- Calling a member function requires specifying the object containing the function

- The calling object is separated from the member function by the dot operator

- Example: `in_stream.open("infile.dat");`



Member Function: Calling Syntax

- Syntax for calling a member function:

```
Calling_object.Member_Function_Name(Argument_list);
```

Errors On Opening Files

- Opening a file can fail for several reasons
  - The file might not exist
  - The name might be typed incorrectly
  - Other reasons

- **Caution**: You may not see an error message if the call to open fails!!
  - Program execution continues!

Catching Stream Errors

- Member function **fail()**, can be used to test the success of a stream operation
  - `fail()` returns a Boolean type (true or false)
  - `fail()` returns true (1) if the stream operation failed

Halting Execution

- When a stream open function fails, it is generally best to stop the program

- The function **exit**, halts a program
  - `exit` returns its argument to the operating system
  - `exit` causes program execution to stop
  - `exit` is NOT a member function

- Exit requires the include and using directives

```
#include <cstdlib>
using namespace std;
```

Using **fail** and **exit**

- Immediately following the call to open, check that the operation was successful:

```
in_stream.open("stuff.dat");
if( in_stream.fail( ) )
  {
          cout << "Input file opening failed.\n";
          exit(1) ;
   }
```

Techniques for File I/O

When reading input from a file do not include prompts or echo the input

- The lines

```
cout << "Enter the number: ";
cin  >> the_number;
cout << "The number you entered is "
     << the_number;
```

become just one line

```
in_file >> the_number;
```

- The input file must contain just the data that's expected

Appending Data

- Output examples we've given so far *create new files*
  - If the output file already contained data, that data is now lost

- To ***append*** new output to the end an existing file use the constant **ios::app** defined in the **iostream** library:

```
outStream.open("important.txt", ios::app);
```

- If the file does not exist, a new file will be created

- Other member functions include those that return where in the output file (or input file) the next data will be
  - Helps with customizing read and writing files
  - To be used carefully!

DEMO!

File Names as Input

- Program users can also enter the name of a file to use for input or for output

- Program name must use a "string of characters" variable
  - You can limit the size of a string by declaring
    a sequence (an array) of characters
  - Declaring a variable to hold a string of characters:

```
char    file_name[16];
```

    - file_name is the name of a variable
    - Brackets enclose the maximum number of characters + 1
    - The variable file_name contains up to 15 characters

- **Note:** Program names cannot take string type variables!
  - This is mostly for legacy reasons with older versions of C++
  - There is a work-around using the function c_str() in the string class
    - Ignore for now…

TO DOs

- Homework #10 due Tuesday 11/1
- Lab #5
  - Due Friday, 10/28, at noon
- Lab #6
  - Will be posted at the end of the weekend

</LECTURE>