# Debug Essentials
# Numerical Base Conversions
# Introduction to Strings in C++

**CS 16: Solving Problems with Computers I**
**Lecture #9**

Ziad Matni
Dept. of Computer Science, UCSB

# Outline

- Debugging your code
  - *Ch. 5.4, 5.5 in the textbook*
- Binary Numbers
- Introduction to Strings and I/O Streams in C++

# Announcements

- **Homework #8 due today**
- Homework #9 is out
- **Don't forget your TAs' and Instructor's office hours!! ☺**

# 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

# 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 w/ it!!!

# 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;
}
```

# Pre- and Post-Conditions

*Concepts of pre-condition and post-condition in functions*

**Pre-condition: What must "be" before you call a function**

- States what is assumed to be true when the function is called
- Function should not be used unless the precondition holds

**Post-condition: What the function will do once it is called**

- Describes the effect of the function call
- Tells what will be true after the function is executed (when the precondition holds)
- If the function returns a value, that value is described
- Changes to call-by-reference parameters are described

# Why use Pre- and Post-conditions?

- Pre-conditions and post-conditions should be the first step in designing a function
- Specify what a function should do BEFORE designing it
  - This minimizes design errors and time wasted writing code that doesn't match the task at hand
- Read textbook's "Supermarket Pricing" case study
  - Ch. 5, from pg. 276 – 281

# Note: Functions Calling Functions

- A function body may contain a call to another function
- The called function declaration must still appear before it is called
- Functions **cannot be defined** in the body of another function

```
void order (int&, int&);
void swap_values (int&, int&);


int main () {
…
…
    order (a, b);
…
…
    return 0;   }


void order(int&  n1, int&  n2) {
    if (n1 > n2)
    swap_values(n1, n2);  }


void swap_values(int& n1, int& n2) {
    int temp = n2;
    n2 = n1;
    n1 = temp;  }
```

# Numerical Conversions in CS

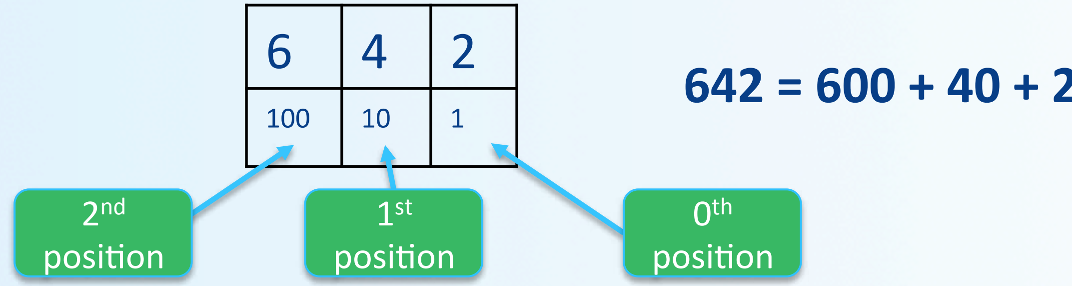
# Counting Numbers in Different Bases

- We "normally" count in 10s
  - Base 10: decimal numbers
  - Number symbols are 0 thru 9
- Computers count in 2s
  - Base 2: binary numbers
  - Number symbols are 0 and 1
  - Represented with **1 bit** ($2^1 = 2$)
- Other convenient bases in computer architecture:
  - Base 8: octal numbers
  - Number symbols are 0 thru 7
  - Represented with **3 bits** ($2^3 = 8$)
  - Base 16: hexadecimal numbers
  - Number symbols are 0 thru F
    - A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
  - Represented with **4 bits** ($2^4 = 16$)
  - **Why are 4 bit representations convenient???**

# Natural Numbers

Counting **642** as 600 + 40 + 2
is counting in TENS (aka BASE 10)

There are 6 HUNDREDS
There are 4 TENS
There are 2 ONES



# Positional Notation in Decimal

**Continuing with our example…**

**642 in base 10 *positional notation* is:**

$$
\begin{aligned}
& 6 \times 10^2 = 6 \times 100 = 600 \\
+ & 4 \times 10^1 = 4 \times 10 = 40 \\
+ & 2 \times 10^0 = 2 \times 1 = 2 \quad = 642 \text{ in base } 10
\end{aligned}
$$

# Positional Notation

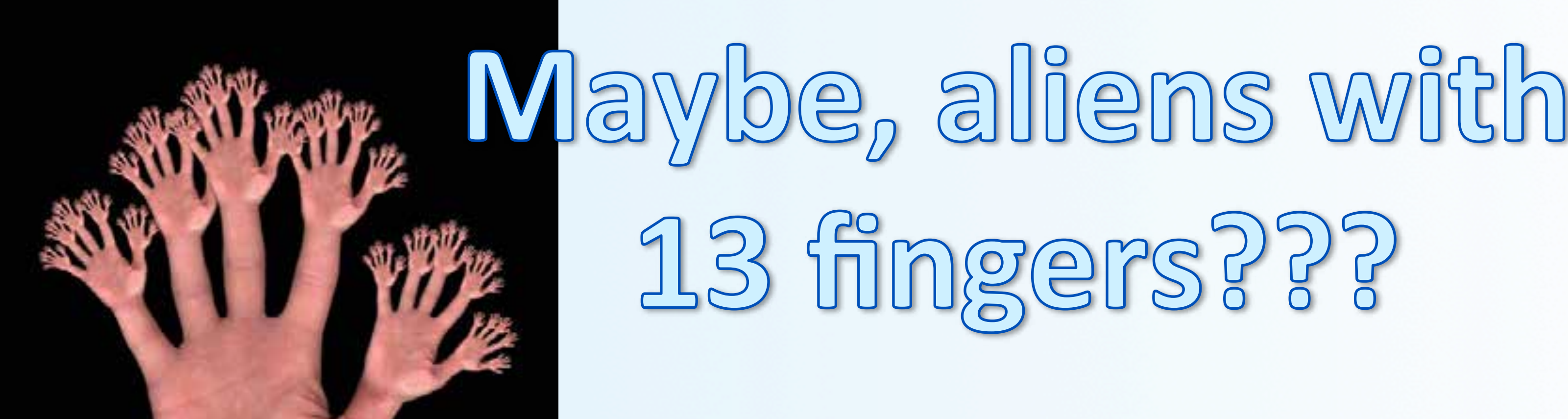
**Anything → DEC**

*What if "642" is expressed in the base of 13?*

$$
\begin{aligned}
& 6 \times 13^2 = 6 \times 169 = 1014 \\
+\ & 4 \times 13^1 = 4 \times 13 = 52 \\
+\ & 2 \times 13^0 = 2 \times 1 = 2 \\
& = 1068 \text{ in base } 10
\end{aligned}
$$

**So, "642" in base 13 is equivalent to "1068" in base 10**

# BUT WHO COUNTS IN BASE 13???!?!?



# COMPUTERS ARE DIGITAL (Binary) MACHINES

# Positional Notation in Binary

**11011** **in base 2** ***positional notation*** **is:**

$$
\begin{aligned}
& 1 \times 2^4 = 1 \times 16 = 16 \\
+ & 1 \times 2^3 = 1 \times 8 = 8 \\
+ & 1 \times 2^2 = 1 \times 4 = 4 \\
+ & 0 \times 2^1 = 1 \times 2 = 0 \\
+ & 1 \times 2^0 = 1 \times 1 = 1
\end{aligned}
$$

So, **1011** in base 2 is 16 + 8 + 0 + 2 + 1 = **27** in base 10

# Converting Binary to Decimal

*Q: What is the decimal equivalent of the binary number **1101110**?*

*A: Look for the position of the digits in the number.*
*This one has 7 digits, therefore positions 0 thru 6*

| **1** | **1** | **0** | **1** | **1** | **1** | **0** |
|---|---|---|---|---|---|---|
| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

**$1 \times 2^6 = 1 \times 64 = 64$**
**$+ 1 \times 2^5 = 1 \times 32 = 32$**
**$+ 0 \times 2^4 = 0 \times 16 = 0$**
**$+ 1 \times 2^3 = 1 \times 8 = 8$**
**$+ 1 \times 2^2 = 1 \times 4 = 4$**
**$+ 1 \times 2^1 = 1 \times 2 = 2$**
**$+ 0 \times 2^0 = 0 \times 1 = 0$**
**= 110 in base 10**

# Other Relevant Bases

- In Computer Science/Engineering, other binary-related numerical bases are used too.

- OCTAL: Base 8 (note that 8 is $2^3$)
  - Uses the symbols: 0, 1, 2, 3, 4, 5, 6, 7

- HEXADECIMAL: Base 16 (note that 16 is $2^4$)
  - Uses the symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

# Converting Binary to Octal and Hexadecimal

*(or any base that's a power of 2)*

- Binary is 1 bit
- Octal is 3 bits ($2^3 = 8$) *octal is base 8*
- Hexadecimal is 4 bits ($2^4 = 16$) *hex is base 16*

- Use the "**group the bits**" technique
  - Always start from the *least significant digit*
  - Group every 3 bits together for bin → oct
  - Group every 4 bits together for bin → hex

# Converting Binary to Octal and Hexadecimal

- Take the example: ***10100110***

*...to octal:*

| 1 0 | 1 0 0 | 1 1 0 |
|---|---|---|
| ***2*** | ***4*** | ***6*** |

***246 in octal***

*...to hexadecimal:*

| 1 0 1 0 | 0 1 1 0 |
|---|---|
| ***10*** | ***6*** |

***A6 in hexadecimal***

# Converting Decimal to Other Bases

Algorithm for converting number in base 10 to other bases

While (the **quotient** is not zero)

1. Divide the decimal number by the new base
2. Make the remainder the next digit to the left in the answer
3. Replace the original decimal number with the quotient
4. Repeat until your quotient is zero

**EXAMPLE:**

Convert the decimal (base 10) number **79** into hexadecimal (base 16)

79 / 16 = 4 R 15 *(15 in hex is the symbol "F")*

4 / 16 = 0 R 4

The answer is: **4F**

# Converting Decimal into Binary

**Convert 54 (base 10) into binary and hex:**

- 54 / 2 = 27 R **0**
- 27 / 2 = 13 R **1**
- 13 / 2 = 6 R **1**
- 6 / 2 = 3 R **0**
- 3 / 2 = 1 R **1**
- 1 / 2 = 0 R **1**

*Sanity check:*
*110110*
*= 2 + 4 + 16 + 32*
*= 54*

**54 (decimal) = 110110 (binary)**

**= 36 (hex)**

# Strings in C/C++

- Recall: C++ is based on C

- Originally (in C), strings were defined as an "array of characters"
  - Called C-Strings and are "legacy" data types in C++
  - Came with the library <cstring>
  - Contains lots of built-in functions that go with C-Strings

- In C++, we got a new library: **<string>**
- Made improvements over the old "C-String"
  - Library contains another collection of functions that work with Strings, but not C-Strings!

# Why Do We Care About C-Strings??

- Their use STILL crops up in C++
- Recall that command-line arguments, specifically argv[*x*] are defined as:

  **char* [ ]**

- That's a classic definition of a C-String
  - So if we want to use these argv[*x*], we'll have to treat them in a C-String fashion…

# Declaring a String in C++

- You have to include the correct library module with:

```
#include <string>
```

- Declare them (and initialize them) with:

```
string MyString="";
// Note the use of double-quotes!
```

- Since strings are made up of characters, you can index individual characters in strings (starting at position 0):

If `MyString = "Hello!"`

Then `MyString[0] = 'H', MyString[1] = 'e',` etc…

# “ vs ‘

- Double quotes are used exclusively for strings
- Single quotes are used exclusively for characters

- We’ll discuss strings and their related functions in an upcoming lecture…

# 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

# To Dos

- Homework #9
- Lab #5

*Prep for Next Week:*

- TUE: I/O Streams and File I/O
  - Read **Chapter 6** in textbook
- THU: Arrays
  - Read **Chapter 7** in textbook

</LECTURE>

# 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;
```

Variable type

Variable name

# 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

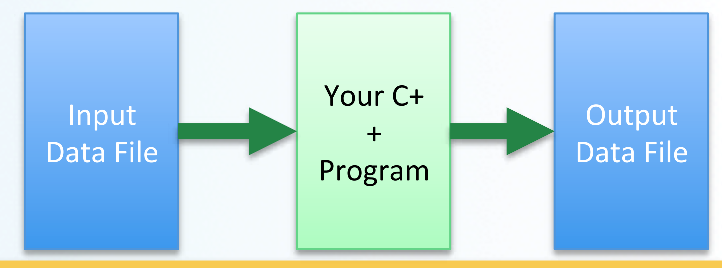
```
#include <fstream>
using namespace std;
```

- Declare an input-file stream variable using

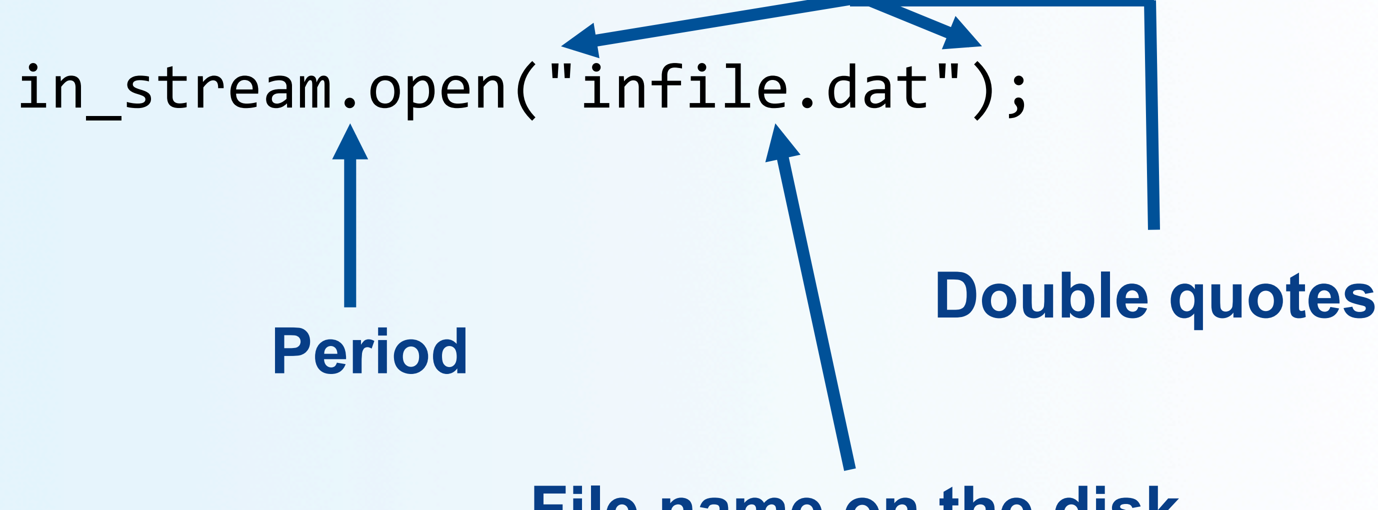
```
ofstream out_stream;
```

Variable name

# 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



# 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!***