Introduction to Programming

Muhammad Waseem Sarwar

Course format/policies

Course Format

- Lectures will be largely interactive. You will be called on randomly to answer questions. Lecture notes typical but not guaranteed.
- 10-minute fairly simple quiz each class just before break.
- Course web page will host lecture notes, quiz answers, homework, general announcements.

http://people.cs.uchicago.edu/~asiegel/courses/cspp50101

Grading

- Breakdown
  - 50% bi-weekly assignments
  - 25% final
  - 25% weekly in-class quizzes
- May share ideas for weekly assignment but must be write up indidvidually.
- Will drop lowest quiz score.

Homework Submission

- Homework due every other Monday before midnight – explicit submission instructions later.
- Please do not ever ask me for an extension.
- Late assignments incur 10% per day penalty up to 3 days. After 3 days, no credit.
- Solutions will be posted after all homeworks are sumbitted (or 3 days, whichever comes first)
- Under special circumstances, you may be excused from an assignment or quiz. Must talk to me ahead of time.

Homework Help

- Very important to use the course newsgroup regularly.
  - ta's/peers will answer questions quickly
  - source of extra credit
- TA's will schedule help sessions, also available by appointment. Will post details on web page.
- If necessary, can schedule appt. with me.
- Homework solutions posted on web page.

Course Strategy

- Not assumed that you have programming experience
- Course will move quickly
- Each subject builds on previous ones
  - More important to be consistent than occasionally heroic
- Start by hacking, back up to cover more fundamental topics

Writing a program

Holistic view

High-level view of programming

- Create new text file
  - this is where instructions the comprise program will be typed
  - this file is typically called you **source code**
  - What this file looks like depends on the choice of **programming language**.
  - As long as you follow synax rules for chosen language, your source code file is valid.
  - In this class we will start with a powerful, venerable, classic language called **C**.

High-level view of programming

- **Compile** the source code.
  - **Compilation** is the process of converting the source code into **machine language** – the very minimalist language that is spoken directly by the machine in use. The machine lanage is stored in a new file.
  - Note: It is possible to program directly in machine language, but it is tedious, ugly, and error-prone.
- **Run** or **execute** the machine-language file.
  - On Unix, this is done by typing the name of the executable file.

Getting Started with C

Getting Started With C

- You will need at least:
  - Computer with an OS (Linux)
  - Text editor (emacs, vi)
  - Compiler   (gcc)
- All of the above suggestions are free in one way or another
  - See http://www.gnu.org
  - See http://www.cygwin.com

Getting Started, Cont.

- These tools are not required, but they are strongly recommended
  - Better for learning
  - Homework must run on Linux gnu compiler
- Important!
  - Become facile with simple Linux and a text editor as quickly as possible
  - Am assuming good knowledge of Linux/emacs

First C Program

A Simple C Program

```c
/* Fig. 2.1: fig02_01.c
   A first program in C */
#include <stdio.h>

int main()
{
   printf( "Welcome to C!\n" );

   return 0;
}
```

```
Welcome to C!
```

- Comments
  - Text surrounded by `/*` and `*/` is ignored by computer
  - Used to describe program
- **`#include <stdio.h>`**
  - Preprocessor directive
    - Tells computer to load contents of a certain file
  - **`<stdio.h>`** allows standard input/output operations

A Simple C Program, Cont.

- `int main()`
  - C programs contain one or more functions, exactly one of which must be `main`
  - Parenthesis used to indicate a function
  - `int` means that `main` "returns" an integer value
  - Braces (`{` and `}`) indicate a block
    - The bodies of all functions must be contained in braces

2.2 A Simple C Program: Printing a Line of Text

- **`Return 0;`**
  - A way to exit a function
  - **`Return 0`**, in this case, means that the program terminated normally

Running the Program on Linux With gcc

- Use emacs, vi, or some other text editor to type in and save program. Good idea to:
  - Name program something meaningful
  - Establish conventions for naming
  - Add a .c suffix to the name
- **Compile** program
  - gcc hello.c [-o whatever]

Running on Linux

- This produces the **executable** named whatever, or a.out by default.
- Type executable name to run.
  - Examples.
    - a.out.
    - whatever.
    - ./a.out.
    - Etc.
- Note: **linker** will be required when our programs become more sophisticated – not necessary now.

Second C Program

User variables, reading user input

```c
/* Fig. 2.5: fig02_05.c
   Addition program */
#include <stdio.h>

int main()
{
   int integer1, integer2, sum;          /* declaration */

   printf( "Enter first integer\n" );   /* prompt */
   scanf( "%d", &integer1 );            /* read an integer */
   printf( "Enter second integer\n" ); /* prompt */
   scanf( "%d", &integer2 );            /* read an integer */
   sum = integer1 + integer2;           /* assignment of sum */
   printf( "Sum is %d\n", sum );        /* print sum */

   return 0;  /* indicate that program ended successfully */
}
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

1. Initialize variables

2. Input

2.1 Sum

3. Print

C Data Types

What do program instructions look like?

- A simple program has at least these three main parts
  - variable declaration
  - variable initialization
  - main body

Variables in Programming

- Represent storage units in a program
- Used to store/retrieve data over life of program
- Type of variable determines what can be placed in the storage unit
- *Assignment* – process of placing a particular value in a variable
- Variables must be *declared* before they are assigned
- The value of a *variable* can change; A *constant* always has the same value

Naming variables

- When a variable is *declared* it is given a name
- Good programming practices
  - Choose a name that reflects the role of the variable in a program, e.g.
    - Good: customer_name, ss_number;
    - Bad : cn, ss;
  - Don’t be afraid to have long names if it aids in readability
- Restrictions
  - Name must begin with a letter; otherwise, can contain digits or any other characters. <u>C is CASE SENSITIVE</u>! Use 31 or fewer characters to aid in portability

Variable Declaration

- All variables must be **declared** in a C program before the first executable statement! Examples:

```
main(){
    int a, b, c;
    float d;
    /* Do something here */
}
```

C Variable Names

- Variable names in C may only consist of letters, digits, and underscores and may not begin with a digit
- Variable names in C are case sensitive
- ANSI standard requires only 31 or fewer characters. Enhances portability to follow this rule
- Should be very descriptive

Variable assignment

- After variables are declared, they must (should) be given values. This is called **assignment** and it is done with the ‘=‘ operator. Examples:

```
float a, b;
int c;
b = 2.12;
c = 200;
```

Basic C variable types

- There are four basic data types in C:
  - char
    - A single byte capable of holding one character in the local character set.
  - int
    - An integer of unspecified size
  - float
    - Single-precision floating point
  - double
    - Double-precision floating point

char variable type

- Represents a single *byte* (8 *bits*) of storage
- Can be *signed* or *unsigned*
- Internally char is just a number
- Numerical value is associated with character via a *character set.*
- ASCII character set used in ANSI C
- Question: what is the difference between:
  - printf(“%c”, someChar);
  - printf(“%d”, someChar);

int variable type

- Represents a signed integer of typically 4 or 8 bytes (32 or 64 bits)
- Precise size is machine-dependent
- Question: What are the maximum and minimum sizes of the following:
  - 32-bit unsigned int
  - 32-bit signed int
  - 64-bit signed/unsigned int
- What happens if these limits are exceeded?

float and double variable types

- Represent typically 32 and/or 64 bit real numbers
- How these are represented internally and their precise sizes depend on the architecture. We won’t obsess over this now.
- Question: How large can a 64-bit float be?
- Question: How many digits of precision does a 64-bit float have?

Additional variable types

- Note that other types can be constructed using the modifiers:
  - short, long, signed, unsigned
- The precise sizes of these types is machine-specific
- We will not worry about them for the time being
- To find out the meaning of short int, etc. on a given system, use <limits.h>

Declaring variables

- All variables must always be *declared* before the first executable instruction in a C program
- Variable declarations are always:
  - var_type *var_name;*
    - int age;
    - float annual_salary;
    - double weight, height; /* multiple vars ok */
- In most cases, variables have no meaningful value at this stage. Memory is set aside for them, but they are not meaningful until assigned.

Assigning values to Variables

- Either when they are declared, or at any subsequent time, variables are assigned values using the “=“ operator.
- Examples

```
int age = 52; //joint declaration/assignment
double salary;
salary = 150000.23;
age = 53; //value may change at any time
```

Assignment, cont.

- Be careful to assign proper type – contract between declaration and assignments must be honored
  - int x=2.13 /* what is the value of x? */
  - double x = 3; /* is this ok? */
  - char c = 300; /* 300 > 1 byte; what happens? */
- General advice
  - Don’t obsess too much over this at beginning
  - Keep it simple, stick to basic data types
  - We will be more pedantic later in the course

Structure of a C program

- So far our C programs are as follows:

```
/* description of program */
#include <stdio.h>
/* any other includes go here */
int main(){
 /* program body */
return 0;
}
```

- Let's learn more about the structure of "program body"

Program Body - declarations

- Always begins with <u>all</u> variable declarations. Some examples:

```
int a, b, c;  /* declare 3 ints named a,b,c */
int d, e;     /* similar to above in two steps */
int f;
int g = 1, h, k=3;
double pi = 3.1415926;
```

- <u>Reading note</u>: K&R mentions that integers can be assigned in *octal* and *hexadecimal* as well. We will discuss this later. Certainly, not important for most applications.

Statements

- Note: all *statements* end with a semicolon!
  - Statements can (with a few exceptions) be broken across lines or ganged on a single line
- Commas separate multiple declarations
- Blank lines have no effect
- Extra spaces between *tokens* has no effect.
- *Comments* are ignored by the compiler

Program Body – Executable Statements

- *Executable statements* always follow variable declarations/initializations
- *Executable statements* include any valid C code that is not a declaration, ie valid C code to do things like:
  - *"multiply the value of a by 10 and store the result in b"*
  - *"add 1 to the value of j and test whether it is greater than the value of k"*
  - *"store 5.2 in the variable x"* (ie assignment)
  - *"print the value of x,y, and z, each on a separate line"*

The printf Executable Statement

- The only executable statements we’ve seen to this point are
  - Assignments
  - The printf and scanf *functions*
  - Assignment expressions with simple operators (+, -)
- Very hard to write any program without being able to print output. Must look at printf in more detail to start writing useful code.

printf(), cont.

- Sends output to *standard out*, which for now we can think of as the terminal screen.
- General form

  printf(format descriptor, var1, var2, …);
- format descriptor is composed of
  - Ordinary characters
    - copied directly to output
  - Conversion specification
    - Causes conversion and printing of next *argument* to printf
    - Each conversion specification begins with %

Printf() examples

- Easiest to start with some examples
  - printf("%s\n", "hello world");
    - Translated: "print hello world as a *string* followed by a newline character"
  - printf("%d\t%d\n", j, k);
    - Translated: "print the value of the variable j as an integer followed by a tab followed by the value of the variable k as an integer followed by a new line."
  - printf("%f : %f : %f\n", x, y, z);
    - English: "print the value of the floating point variable x, followed by a space, then a colon, then a space, etc.

More on format statements

- The format specifier in its simplest form is one of:
  - %s
    - sequence of characters known as a *String*
    - Not a fundamental datatype in C (really an *array* of char)
  - %d
    - Decimal integer (ie base ten)
  - %f
    - Floating point
- Note that there are many other options. These are the most common, though, and are more than enough to get started.

Invisible characters

- Some special characters are not visible directly in the output stream. These all begin with an escape character (ie \);
  - \n newline
  - \t horizontal tab
  - \a alert bell
  - \v vertical tab
- See K&R p.38 for more details

Arithmetic Operations

- Five simple *binary arithmetic operators*
  1. + “plus” → c = a + b
  2. - “minus” → c = a - b
  3. * “times” → c = a * b
  4. / “divided by” c = a/b
  5. % “modulus” c = a % b

1. What are the values of c in each case above if
   - int a = 10, b = 2;
   - float a = 10, b = 2;
   - int a = 10; float b = 2; ??

Relational Operators

- Four basic operators for comparison of values in C. These are typically called *relational operators:*

  1. > “greater than”
  2. < “less than”
  3. >= “greater than or equal to”
  4. <= “less than or equal to”

1. For the declaration

int a=1,b=2,c;

what is the value of the following expressions?

a > b; a<b; a>=b;a<=b

Relational Operators, cont.

- Typically used with *conditional* expressions, e.g.
  - if (a < 1) then …
- However, also completely valid expressions which evaluate to a result – either 1 (true) or 0 (false).

```
int c, a=2, b=1;
c = (a > b)
```

What is the value of c?

- Note: We'll talk about *order of precedence* for multipart expressions a little later. For now, we force an order using parentheses.

Equality Operators

- C distinguished between relational and *equality operators.*
- This is mainly to clarify rules of order of precedence.
- Two equality operators

1. == "is equal to"
2. != "is not equal to"

- These follow all of the same rules for relational operators described on the previous slide.

Logical Operators

- *Logical Operators* are used to create compound expressions
- There are two logical operators in C

1. || "logical or"
   - A compound expression formed with || evaluates to 1 (true) if any one of its components is true

2. && "logical and"
   - A compound expression formed with && evaluates to true if all of its components are true

Logical Operators, cont.

- Logical operators, like relational operators, are typically used in conditional expressions

1. if ((a == 1) && (b < 3) || (c == 1)) etc.

- However, these can also be used in regular expressions

```
int a = 1, b = 2, c = 3, d;
d = ( a > b ) || ( c == (b – 1) );
```

What is the value of d here?

Reading keyboard input

- To be useful, program must be able to read data from external source, e.g.
  - User input from keyboard
  - Database
  - File
  - Socket
- In next slide we cover the scanf library function. It is like printf but reads user-typed input rather than prints it.

Scanf function

- In <stdio.f>, so no new #include(‘s)
- Basic syntax
  - scanf(format-specifier, &var1, &var2, etc.);
  - Format-specifier is identical to printf
  - We do not need to understand everything here, just enough to do some basic I/O
- Examples
  - int a; scanf(“%d”,&a);
  - double x; scanf(“%f”,&x);
- Blocks program until user enters input!

Another technique for passing data from the keyboard

- main() can also be written as

main(int argc, char *argv[])

- If main is written in this way, information can be passed directly from the keyboard to the program at the time of execution
  - For example, we may run a program called a.out as: PROMPT > a.out Andrew Siegel
  - When a.out is run the two tokens Andrew and Siegel are passed into the program and can be obtained by querying argv and argc
  - Note: this involves some concepts that go beyond what we have learned so far. We will understand fully later.

Passing data, cont.

- When this technique is used, each token is stored as a separate *element* in the *array* argv
- The first token passed to the program is stored in argv[1], the second token in argv[2], etc.
- argc stores the (integer) number of tokens passed in
- A simple example will clarify this

argc/argv example

- int main (int argc, char* argv[]){

```
    printf(“%s %d %s \n”, “you entered”, argc, “arguments”);
    printf(“%s: %s\n”, “the zeroth arg is the program name”, argv[0]);
    printf(“%s: %s\n”, “the first argument is”, argv[1]);
    printf(“%s: %s\n”, “the second argument is, argv[2]);
}
```

```
> gcc argv_example.c –o argv_example
> argv_example hello world
  you entered 3 arguments
  the zeroth argument is the program name: argv_example
  the first argument is hello
  the second argument is world
```

argc/argv cont.

- Note that to do this completely generally we would need to use a *while* or *for loop*
- We will study while loops shortly
- Also note that argv reads all arguments as Strings (ie sequences of characters). So, we cannot simply pass two number and add them, for example. First, we would have to convert to a numeric type.

Converting String to integer

- An important function when using argv/argc is atoi.
- atoi converts a String argument to an integer in the following way:

```
int input, output;
input = atoi(argv[1]);
output = input + 1;
printf("%d\n", output);
> a.out 333
  334
```

Reading single characters

- Another important pair of functions for keyboard input/output is getchar/putchar
- getchar reads a single character from the *input stream*; putchar write a single character to the standard out for example:

```
 int c;
c = getchar();  /* blocks until data is entered
                   if more than one char is entered only first is read */
putchar(c);     /* prints value of c to screen */
```

While loops

- A simple technique for repeating a statement or group of statements until some specified condition is met
- General form:

```
while (expr){
   statement1;
   statement2;
          .
          .
 }
```

- If *expr* evaluates to true (ie not 0), then perform statement1, etc. Otherwise, skip to end of while block.
- Repeat until *expr* evaluates to false (ie 0).

While example

```
/* a program to loop over user input and print to screen */
#include <stdio.h>
int main(int argc, char* argv[]){
  int counter;           /* declarations */
  counter = 1;           /* executable body */
  while (counter < argc){
      printf("%s %d: %s\n", "argument number", counter, argv[counter]);
      counter = counter + 1;  /* equivalent to counter++ or ++counter */
  }
   return 0;
}
```

If example

```
/* a program to loop over user input and print back to screen
   with a little error checking */
int main(int argc, char *argv[]){
  int counter = 1;
/* check to make sure the user entered something */
 if (argc < 2){
     printf("%s\n", "error; must enter at least one argument!");
     exit(1);         /* exit(1) will end the program */
   }
/* if ok, continue as before */
while (counter < argc){
      printf("%s %d: %s\n", "argument number", counter, argv[counter]);
      counter = counter + 1;  /* equivalent to counter++ or ++counter */
  }
}
```

Getchar/putchar example

```
/* uses getchar with while to echo user input */
#include<stdio.h>
int main(){
  int c;              /* holds the input character value */
  c = getchar(); /* reads first character from input stream
                        with keyboard, this is signaled by Enter key*/
  while (1){       /* loop forever */
    putchar(c);  /* write char to keyboard */
    c = getchar(); /*get next char in stream */
  }
}
```

Input redirection

- Files can be sent to an input stream by using the unix redirection command '<'. For example, if we wish to pass input into a program call process_text, we can do:

```
process_text < somefile.txt
```

where somefile.txt is a text file that exists in the current directory. This sends the contents of somefile.txt into process_text as standard input

What Is a Computer?

- Computer
  - Device capable of performing computations and making logical decisions
  - Computers process data under the control of sets of instructions called computer programs

What Is a Computer, Cont.

- Hardware
  - Various devices comprising a computer
  - Keyboard, screen, mouse, disks, memory, CD-ROM, and processing units
- Software
  - Programs that run on a computer1.3 computer organization

History of C

- C
  - Evolved by Ritchie from two previous programming languages, BCPL and B
  - Used to develop UNIX
  - Used to write modern operating systems
- Hardware independent (portable)
  - By late 1970's C had evolved to "traditional C"

History of C

- Standardization
  - Many slight variations of C existed, and were incompatible
  - Committee formed to create a "unambiguous, machine-independent" definition
  - Standard created in 1989, updated in 1999

Language Types

- Three types of programming languages
  1. Machine languages
     - Strings of numbers giving machine specific instructions
     - Example:

```
+1300042774
+1400593419
+1200274027
```

  2. Assembly languages
     - English-like abbreviations representing elementary computer operations (translated via assemblers)
     - Example:

```
Load   BASEPAY
Add    overpay
Store  GROSSPAY
```

Language Types, Cont.

3. High-level languages
   - Codes similar to everyday English
   - Use mathematical notations (translated via compilers)
   - Example:

```
grossPay = basePay + overTimePay
```

High-level Languages

- “high-level” is a relative term
- C is a relatively low-level high-level language
- Pascal, Fortran, COBOL are typical high-level languages
- Java, Python, Perl, VB are examples of high-level high-level languages
- Application specific languages (Matlab, Javascript, VBScript) are even higher-level.

C Programming Language

- What is C?
  - C is a structured, relatively low-level, portable programming language.
- Why study C?
  - Many popular software tools are written in C.
  - Has strongly influenced many other languages.
    - C-shell, java, C++, Perl, etc.
  - Forces the user to understand fundamental aspects of programming.
  - Very concise language.

C, cont.

- Is C object-oriented?
  - No. C++ (its successor) is.
- Can a non OO language be useful?
  - Yes.
- Is C a hard language to learn?
  - No, but it does take a little getting used to.
- What is the ANSI part?
  - American national standards institute – uniform standard definition of C for portability.

C Data Types

- There are only a few basic data types in C:
  - char
  - int
  - float
  - double
- **short**, **long**, **signed** and **unsigned** are additional qualifiers.
  - will discuss later

C Data types, cont.

- char
  - A single byte (capable of holding one character in the local character set).
- int
  - An integer, typically reflecting the natural size of integers on the host machine.
- float
  - Single precision floating point (typically 8-bit)
- double
  - Double precision floating point (typicall 4-bit)

- Suggested assignment:
  - Using what you know so far, together with appendix B11 of K&R, determine what the variable sizes are on your platform.
  - Note: you'll need to include the <limits.h> header file.