CS3157: Advanced Programming

Lecture #6
Oct 3
Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

• Feedback
• Intro to C
  – Background
  – Compiling
  – Basic data structures
  – Basic I/O
  – Types conversion
  – Loops
  – Branching
Feedback

• Generally good pace of labs
• Complaints about lab workload
  – 4 credit course
  – Will make homework projects / class knowledge easier in the long run
  – Am trying to balance learning and amount of effort, hard to sometime gauge effort
  – Feedback essential

Roadmap

• How this all fits together
  – We covered perl (duct-tape programming)
  – CGI programming USING perl

  – Will now move to c, which is a more low level programming language
  – Will learn to work with c, and then CGI+c
  – Then CGI+perl+c etc
  – Get the best of any programming language in a project
Why Learn C?

• C provides stronger control of low-level mechanisms such as memory allocation, specific memory locations

• C performance is usually better than Java and usually more predictable (very task dependant)

Why Learn c continued

• Java hides many details needed for writing code, but in C you need to be careful because:
  – memory management responsibility left to you
  – explicit initialization and error detection left to you
  – generally, more lines of (your) code for the same functionality
  – more room for you to make mistakes

• most older code is written in C, will need it if upgrading or interfacing
Background

C
– Dennis Ritchie in late 1960s and early 1970s
– systems programming language
– make OS portable across hardware platforms
– not necessarily for real applications—could be written in Fortran or PL/I

Background II

C++
– Bjarne Stroustrup (Bell Labs), 1980s
– object-oriented features

Java
– James Gosling in 1990s, originally for embedded systems
– object-oriented, like C++
– ideas and some syntax from C
Background III

• C is early-70s, procedural language

• C advantages:
  – direct access to OS primitives (system calls)
  – more control over memory
  – fewer library issues—just execute

• C disadvantages:
  – language is portable, but APIs are not
  – no easy graphics interface
  – more control over memory (i.e., memory leaks)
  – pre-processor can lead to obscure errors

C vs Java

• Java program
  – collection of classes
  – class containing main method is starting class
  – running java StartClass invokes StartClass.main method
  – JVM loads other classes as required

• C program
  – collection of functions
  – one function – main() – is starting function
  – running executable (default name a.out) starts main function
  – typically, single program with all user code linked in— but can be
dynamic libraries (.dll, .so)
Example

• Java

```java
public class hello {
    public static void main( String[] args ) {
        System.out.println( "hello world! ");
    }
}
```

• C

```c
#include <stdio.h>
int main() {
    printf( "hello world! ");
    return 0;
}
```

- `#include <stdio.h>` to include header file `stdio.h`
- `#` lines processed by pre-processor
- No semicolon at end of pre-processor lines
- Lower-case letters only—C is case-sensitive
- `int main() { ... }` is the only code executed
- `printf( " /* message you want printed */ " );`
- `\n` = newline, `\t` = tab
- `\` in front of other special characters
C vs Java …Running

- Java programs are compiled and interpreted:
  - javac converts foo.java into foo.class
  - class file is not machine-specific— it is byte code
  - byte code is then interpreted by JVM
  - and each JVM is machine-specific

- C programs are compiled into object code and then linked into executables
  (to allow for multiple object files and libraries to be compiled together into one program):
  - gcc compiles foo.c into foo.o and then links foo.o into a.out
  - you can skip writing foo.o if there is only one object file used to create your executable
  - a.out is executed by OS and hardware
  - the C compiler is machine-specific, creating code that executes on specific OS/hardware

Compiling c

- gcc is the C compiler we’ll use in this class
- it’s a free compiler from Gnu (i.e., Gnu C Compiler)
- gcc translates C program into executable for some target
- default file name a.out
- behavior of gcc is controlled by command-line switches
  $ gcc hello.c
  $ a.out
  hello world!
Compiling your program

two-stage compilation
1. pre-process and compile: gcc -c hello.c
2. link: gcc -o hello hello.o

linking several modules:
>gcc -c a.c -> a.o
>gcc -c b.c -> b.o
>gcc -o hello a.o b.o

using a library, for example the "math" library (libm):
>gcc -o calc calc.c -lm

Compiling problems

• errors can come from multiple sources:
  – pre-processor: missing include files
  – parser: syntax errors
  – assembler: rare
  – linker: missing libraries and references
  – e.g., undefined names will be reported when linking:
    undefined symbol first referenced in file
    _print program.o
    ld fatal: Symbol referencing errors
    No output written to file.

• if gcc gets confused, there can be hundreds of messages!
  – fix first message first, and then retry—ignore the rest
• gcc will produce an executable with warnings
• gcc is more forgiving than javac!
c pre-processor

- the C pre-processor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it
  - pre-processor directives start with # at beginning of line
- used to:
  - include files with C code (typically, “header” files containing definitions; file names end with .h)
  - define new macros (later – not today)
  - conditionally compile parts of file (later – not today)
- gcc -E shows output of pre-processor
- can be used independently of compiler

pre-processor II

- file inclusion
  #include "filename.h"
  #include <filename>
  - inserts contents of filename into file to be compiled
  - "filename.h" relative to current directory
  - <filename> relative to /usr/include or in default path (specified by -I compiler directive); note that file is named verb+filename.h+
- import function prototypes (in contrast with Java import) (more about function prototypes later)
- examples:
  #include <stdio.h>
  #include "mydefs.h"
  #include "/home/shlomo/programs/defs.h"
Comments

1. /* any text until this */

2. // until end of line

• convention for longer comments:
  /*
   * AverageGrade()
   * Given an array of grades, compute the average.
   */

• avoid **** boxes - hard to edit, usually look ragged.

Data Types

• Very important when trying to resource memory/cpu
  • float has 6 bits precision
  • double has 15 bits precision
  • Range can change depending on machine type, generally int is native to the machine type

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
</tr>
</tbody>
</table>
Types II

- unsigned char
- unsigned short
- unsigned int
- unsigned long

- Byte size is the same, but can now have greater range
- /usr/include/limits.h

Library

- Access libraries using the include statement
- Generally include the header file
- Compiler links them automatically
- Example:
  - stdio.h
  - Try:
    - man stdio
stdio.h

- Access stdio functions by
  - using #include <stdio.h>
  - compiler links it automatically

- defines stdin, stdout, stderr
- use for character, string and file I/O (later)

- printf
  - %[flags][width][.precision][modifiers]type
stdio.h : printf, type specifier

- int printf(const char *format, ...) formatted output to stdout

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>d or i</td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td>e</td>
<td>Scientific notation (mantise/exponent) using e character</td>
</tr>
<tr>
<td>E</td>
<td>Scientific notation (mantise/exponent) using E character</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point</td>
</tr>
<tr>
<td>g</td>
<td>Use shorter %e or %f</td>
</tr>
<tr>
<td>G</td>
<td>Use shorter %E or %f</td>
</tr>
<tr>
<td>o</td>
<td>Signed octal</td>
</tr>
<tr>
<td>s</td>
<td>String of characters</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal integer</td>
</tr>
<tr>
<td>X</td>
<td>Unsigned hexadecimal integer (capital letters)</td>
</tr>
<tr>
<td>p</td>
<td>Address pointed by the argument</td>
</tr>
<tr>
<td>n</td>
<td>Nothing printed. The argument must be a pointer to integer where the number of characters written so far will be stored.</td>
</tr>
</tbody>
</table>

printf flags

- %[flags][width][.precision][modifiers]type

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left align within the given width. (right align is the default).</td>
</tr>
<tr>
<td>+</td>
<td>Forces to preceed the result with a sign (+ or -) if signed type. (by default only - (minus) is printed).</td>
</tr>
<tr>
<td>Blank</td>
<td>If the argument is a positive signed value, a blank is inserted before the number.</td>
</tr>
<tr>
<td>#</td>
<td>Used with o, x or X type the value is preceeded with 0, 0x or 0X respectively if non-zero.</td>
</tr>
<tr>
<td></td>
<td>Used with e, E or f forces the output value to contain a decimal point even if only zeros follow.</td>
</tr>
<tr>
<td></td>
<td>Used with g or G the result is the same as e or E but trailing zeros are not removed.</td>
</tr>
</tbody>
</table>
example

```c
int class_size = 35;
char *class_name = "3157 adv prog";

printf("Welcome to our test program\n");

printf("the %s class size is %d",
       class_name, class_size);
```

**stdio.h: scanf**

- int scanf(const char *format, ...) formatted output to stdout
Example: scanf/printf

#include <stdio.h>
void main( void ) {
    int n = 0; /* initialization required */
    printf( "how much wood could a woodchuck chuck\n" );
    printf( "if a woodchuck could chuck wood?" ); /* prompt user */
    scanf( "%d", &n ); /* read input */
    printf( "the woodchuck can chuck %d pieces of wood!\n", n );
    return;
}

output

$ a.out
how much wood could a woodchuck chuck
if a woodchuck could chuck wood? 12345
the woodchuck can chuck 12345 pieces of wood!
Loops

- loops in C are just like in Java
- there are 2 methods for looping:
  - counter-controlled (loop for a fixed number of times)
  - sentinel-controlled (loop while a condition is true)
- there are 3 statements for implementing the 2 methodologies:
  - for
  - while
  - do...while
- as always: beware the infinite loop!
- Ctrl-C interrupts your executing C program
- exercise: can you write 6 loops, one for each method-statement combination?

Branching

- branching in C is just like in Java
- there are 2 ways to do branching:
  - if/else
  - switch
- questions:
  - which is more flexible and powerful?
  - one can always be translated into the other, but not the other way around— which is which?
For next time:

• Lab on Wednesday
• For anyone observing Jewish new year, I will have extra lab hours Thursday 2-4.