CS3157: Advanced Programming

Lecture #7
Feb 20

Shlomo Hershkop
shlomo@cs.columbia.edu

1

Outline

- Working with C
  - Compiling
  - Basic data structures
  - Basic I/O
  - Types conversion
  - Loops
  - Branching
  - compiling
  - Control flow
  - Arrays
  - Pointers
  - strings
  - string library
  - string tokenizing
  - Memory allocation intro
- Reading
  - K & R skim 1-3.
  - Read: K & R 4.1-4.3, 7.1-7.5)

2

Announcements

- Please do the reading
- Will get lost quickly if not ☺
- Don’t worry if you don’t have any experience

3

Brief Overview

- For the c section of the course, here are some tips

1. Write your course code
2. Try to compile
3. Debug compile bugs, goto step 1
4. Try step 2 again
5. Run debugger to catch run time bugs
6. Run memory profiler to catch memory bugs
7. Have running product
8. Add one last cool feature and jump to step 3 ☺

4

How to make your c code run

- 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 machine platform
- default file name a.out
- behavior of gcc is controlled by command-line switches
- Will create files to help in compiling out programs

$ gcc hello.c

$. a.out

hello world!

5

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
    == hello

using a library, for example the "math" library (libm):

>gcc -o calc calc.c -lm

6

C control flow

- blocks are enclosed in curly brackets
- functions are blocks
- main() is a function
- blocks have two parts:
  - variable declaration (“data segment”)
  - code segment
- in C, variables have to be declared before they are used
- initializations can occur at the end of the declaration section, but before the code section

7

Break down of running program

- A program is a collection of functions
- The function names main is launched first

8

First c program

```
/* First c program */

int main(void){

  printf("Hello Everyone\n”);

  return 0;
}
```

9

compile

- gcc –o test simple.c
- ./test

10

Steps to running program

- Write code
  - Platform independent (for the most part)
- Preprocess the code
  - Understand and reinterpret parts
- Compile the code generate object files
  - Turn it into machine code, use optimizers
- Link object files to executable
- Load executable to running code

11

Split personalities

- In c and cpp normal to divide definition of code (header files .h) and working code (.c files)

- So will have function declaration
- `int foo();`
- And function definitions
- `int foo(){. . . . . }`

12

A macro

- A macro is a section of code, which has been given a name
- When you use the name, the preprocessor will replace it with the code contents

13

- #define BUFFER_SIZE 1024
- Convention to use upper case
- Will be replaced exactly with the stuff after the name
- int x = BUFFER_SIZE;

14

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 for the compiler
  - 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
  - conditionally compile parts of file (later – not today)
- gcc -E shows output of pre-processor
- can be used independently of compiler

15

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) examples:

```
#include <stdio.h>
#include "mydefs.h"
#include "/home/shlomo/programs/defs.h"
```

16

Comments

/* any text until this */

- convention for longer comments:

```
/*
* AverageGrade()
* Given an array of grades, compute the average.
*/
```

- Don’t get carried away with comment boxed
- **** boxes - hard to edit, usually look ragged.

17

Where to begin?

- Lets talk about what are the primitive data types:

18

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

Type	Bits
char	8
short	16
int	32
long	32
float	32
double	64

19

Types II

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

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

20

Use in functions

- Variables must be declared in the beginning of the function to be used
- Common mistake: forgetting to declare at top of function

21

Intro arrays

- An array is a group of memory locations with the same name and type
- To get to a particular element in the array we need
  - data type
  - name
  - Length or position
- Array length can be determined:
  - statically— at compile time (when we code)
    - e.g., char str1[10];
  - dynamically— at run time (more on this later)
    - e.g., char *str2;

22

- defining a variable is called “allocating memory” to store that variable
- defining an array means allocating memory for a group of bytes,
- individual array elements are indexed
  - starting with 0
  - ending with length -1
- indices follow array name, enclosed in square brackets ([ ])

  e.g., name[25]

23

- Initializing the arrays are your problem

int a[3];

….

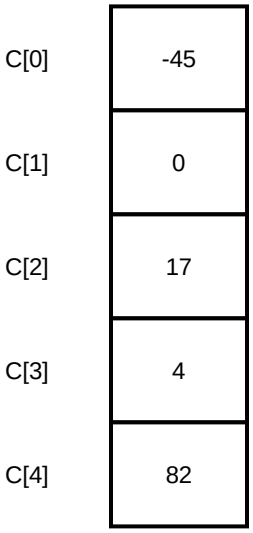
X = a[1]; ……

- Bound checking is your problem

printf(“%d”,a[100]); …..

24

int C[5]



We can say for example

X = C[4] / 5;

Declarations:

int b[100],v[3];

25

More arrays

- Can also create arrays in the following manners

1. int a[] = {1,2,3};
2. int b[3] = {6,3,7};
3. int n[10] = {0};

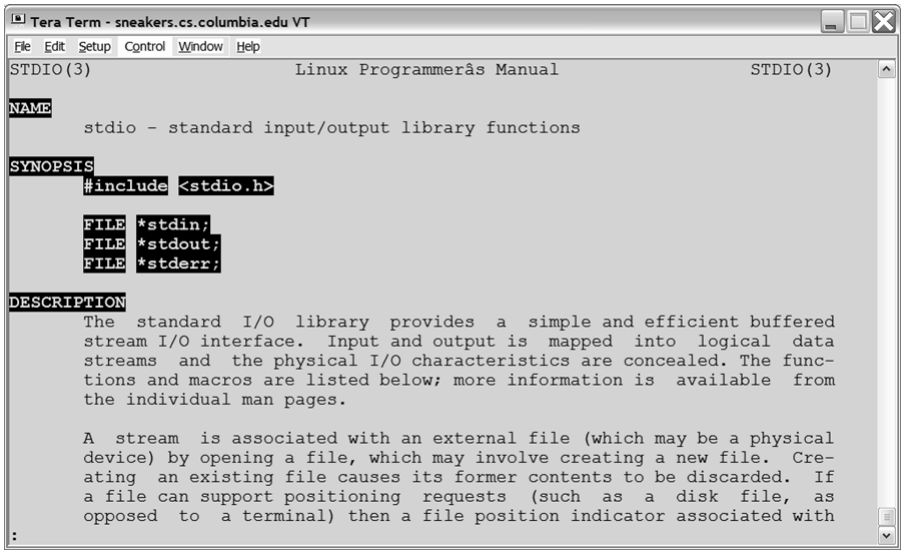
Note you need to initialize the array elements, 3 is a trick case.

26

Library

- Access libraries using the include statement
- Generally include header files
- Compiler links them automatically
- Example:
  - Standard input/output: stdio.h
  - To look up information use the man page: `man stdio`

27



28

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

29

printf

- The way printf works is it takes a format to print out and then the data to add to the format
- One or more of the following:
  - %[flags][width][.precision][modifiers]type
  - “%d”
    - Means a single number
  - “%d %d %d”
    - ??

30

- printf (“%d %d”,a,b);

31

stdio.h : printf, type specifier

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

c	Character	a
d or i	Signed decimal integer	392
e	Scientific notation (mantise/exponent) using e character	3.9265e2
E	Scientific notation (mantise/exponent) using E character	3.9265E2
f	Decimal floating point	392.65
g	Use shorter %e or %f	392.65
G	Use shorter %E or %f	392.65
o	Signed octal	610
s	String of characters	sample
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (capital letters)	7FA
p	Address pointed by the argument	B800:0000
n	Nothing printed. The argument must be a pointer to integer where the number of characters written so far will be stored.	

32

printf flags

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

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

33

example

```
int class_size = 35;
char class_name[15] = “3157 adv prog”;

printf(“Welcome to our test program\n”);


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

34

int array

```
1.  #include <stdio.h>
2.  #define MAX 6

3.  int main( void ) {
4.  int arr[MAX] = { -45, 6, 0, 72, 1543, 62 };
5.  int i;

6.  for ( i=0; i<MAX; i++ ) {
7.      printf( “[%d] = %d \n", i, arr[i] );
8.  }

9.  } /* end of main() */
```

35

stdio.h: scanf

- `int scanf(const char *format, ...)`

36

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;
}
```

37

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!
```

38

Loops

- loops in C are just like in Java
- there are 2 methods for looping:
  - counter-controlled (loop for a fixed number of times)
  - sentinal-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

39

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?

40

Pointer power

- Variables that contain memory addresses as their values
- Data types we've learned about in C use direct addressing
- Pointers facilitate indirect addressing
- Declaring pointers:
  - pointers indirectly address memory where data of the types we've already discussed is stored (e.g., int, char, float, etc.)
  - declaration uses asterisks (*) to indicate a pointer to a memory location storing a particular data type
  - Called dereferencing a pointer
- example:

```
int *count;
float *avg;
```

41

Pointers: nitty gritty

- ampersand & is used to get the address of a variable (dereference a pointer)

- example:

int count = 12;

int *countPtr = &count;

- &count returns the address of count and stores it in the pointer variable countPtr

42

Another example

- here's another example:

```
int i = 3, j = -99;
int count = 12;
int *countPtr = &count;
printf ( "%d", *countPtr);
```

- Here is the memory picture:

43

Arrays as pointers

- an array is some number of contiguous memory locations
- an array definition is really a pointer to the starting memory location of the array
- and pointers are really integers
- so you can perform integer arithmetic on them
- e.g., +1 increments a pointer, -1 decrements
- you can use this to move from one array element to another

44

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
int i, *j, arr[5];
srand( time ( NULL ));
for ( i=0; i<5; i++ )
arr[i] = rand() % 100;
printf( "arr=%p\n",arr );
for ( i=0; i<5; i++ ) {
printf( "i=%d arr[i]=%d &arr[i]=%p\n",i,arr[i],&arr[i] );
}
j = &arr[0];
printf( "\nj=%p *j=%d\n",j,*j );
j++;
printf( "after adding 1 to j:\n j=%p *j=%d\n",j,*j );
}
```

45

Output

```
arr=0xbffff4f0
i=0 arr[i]=29 &arr[i]=0xbffff4f0
i=1 arr[i]=8 &arr[i]=0xbffff4f4
i=2 arr[i]=18 &arr[i]=0xbffff4f8
i=3 arr[i]=95 &arr[i]=0xbffff4fc
i=4 arr[i]=48 &arr[i]=0xbffff500
j=0xbffff4f0 *j=29
after adding 1 to j:
j=0xbffff4f4 *j=8
```

46

Pointer operations

- Difference between
  - ptr++
  - *ptr++
- int b[5] ….
  int *bPtr;

  bPtr = b //or
  bPtr = &b[0]

47

- Careful when moving pointers
- bPTr += 2;

  the memory location isn't simply incremented by 2…..depends on size of type being pointed to.

48

Strings

- storing multiple characters in a single variable
- data type is still char
- BUT it has a length
- last character the is terminator: ’\0’, aka NULL
- string constants are surrounded by double quotes: "
- example:
  - char s[6] = "ABCDE";

49

String II

- char s[6] = “ABCDE”;
- Memory storage looks like:

A	B	C	D	E	\0

- Need to remember that you are really accessing indices 0 – (*length*-2) since the value at *length*-1 is always \0

50

Using strings

- printing strings
- format sequence: %s
- example:

```
#include <stdio.h>
int main() {
char str[6] = "ABCDE";
printf( "str = %s\n", str );
} /* end of main() */
```

51

String Library

- to use the string library, include the header in your C source file:

```
#include <string.h>
```

- string length function:

```
int strlen( char *s );
```

- this function returns the number of characters in s; note that this is NOT the same thing as the number of characters allocated for the string array

- string comparison function:

```
int strcmp( const char *s1, const char *s2 );
```

- "This function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared."

man strcmp

52

copying functions:

```
char *strcpy( char *dest, char
  *source );
```

- copies characters from source array into dest array up to NULL

```
char *strncpy( char *dest, char
  *source, int num );
```

- copies characters from source array into dest array; stops after num characters (if no NULL before that); appends NUL

53

Search

```
char *strchr( const char
  *source, const char ch );
```

- returns pointer to first occurrence of ch in source; NULL if none

```
char *strstr( const char
  *source, const char *search );
```

- return pointer to first occurrence of search in source

54

String Parsing

```
char *strtok( char *s1, const char
  *s2 );
```

- breaks string s1 into a series of tokens, delimited by s2
- called the first time with s1 equal to the string you want to break up
- called subsequent times with NULL as the first argument
- each time is called, it returns the next token on the string
- returns null when no more tokens remain

55

Example

```
char inputline[1024];
char *name, *rank, *serial_num;
printf( "enter name+rank+serial number: " );
scanf( "%s", inputline );
name = strtok( inputline,"+" );
rank = strtok( null,"+" );
serial_num = strtok( null,"+" );
```

56

Formatting functions

`int sscanf(char *string, char *format, ...)`

- parse the contents of string according to format
- placed the parsed items into 3rd, 4th, 5th, ... argument
- return the number of successful conversions

`int sprintf(char *buffer, char *format, ...)`

- produce a string formatted according to format
- place this string into the buffer
- the 3rd, 4th, 5th, ... arguments are formatted
- return number of successful conversions

- format characters are like printf and scanf (see notes from earlier lectures)

57

Memory allocations

- One of the most powerful features of c is the ability of the programmer to create more memory space during the execution of the program.
- Limited by physical machine memory
- If you want to be able to create memory, you also need to free it manually

58

malloc /sizeof / free

- charPtr = malloc (sizeof (…));
- free (charPtr)

59

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!

60

Review for midterm

- Will post online review notes, and some sample questions etc.

- Open book, open notes
- No computers or any computing device
- Will be testing you on understanding of material covered

61

- Should know perl
  - Basic types and usages, comments etc.
  - Perl debugger
  - File handling, Subroutines
  - md5
- CGI
  - What, how, and sometimes why
  - How to use perl/CGI
- C
  - Basics
  - Types/pointers/arrays
  - Linking, compiling, and makefiles

62