cs3157 lecture #3 notes.

mon 3 feb 2003

http://www.cs.columbia.edu/~cs3157

- news
  - homework #1 has been posted
- today's topics
  - introduction to programming in C
  - compiling and the C pre-processor
  - data types
  - basic I/O (stdio library)
  - math library
  - looping
  - branching

intro (1): why learn C after Java?

- C provides better control of low-level mechanisms such as memory allocation, specific memory locations
- C performance is usually better than Java and usually more predictable
- 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

intro (2): history before C.

- 1960's: many new languages
  - COBOL for commercial programming (databases)
  - FORTRAN for numerical and scientific programs
  - PL/I as second-generation unified language
  - LISP for early AI research
  - Assembler for operating systems and timing-critical code
- Bell Labs (research arm of Bell System → AT&T → Lucent) needed own OS
- Ken Thompson: B
- Dennis Ritchie: new language = B + types

intro (3): history of C.

- 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
- 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

intro (4): C vs Java.

- Java is mid-90s, high-level *Object-Oriented (OO)* language
- 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

intro (5): C vs Java.

Java	**C**
object-oriented	function-oriented
strongly-typed	can be overridden
polymorphism (+,==)	very limited (integer/float)
classes for name space	(mostly) single name space, file-oriented
macros are external, rarely used	macros common (pre-processor)
layered I/O model	byte-stream I/O
automatic memory management	function calls (C++ has some support)
no pointers	pointers (memory addresses) common
by-reference, by-value	by-value parameters
exceptions, exception handling	signals, signal handling
concurrency (threads)	library functions (system calls)
length of array	on your own
string as a type	on your own (byte[] or char[] with \0 end)
dozens of common libraries	OS-defined

intro (6): 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)

intro (7): simple example, C vs Java.

Java

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

C

```
#include <stdio.h>
int main( int argc, char *argv[] ) {
  puts( "hello world!" );
  return 0;
}
```

intro (8): dissecting the example.

- `#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
- `void main(void){ ... }` is the only code executed
- `puts( " /* message you want printed */ ");`
- `\n` = newline, `\t = tab`
- `\` in front of other special characters
- `printf( "Have you heard of \"The Matrix\" ? \n" );`

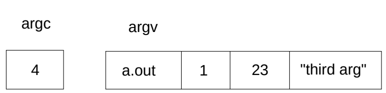
executing C programs (1).

```
int main( int argc, char argv[] )
```

- `argc` is the argument count
- `argv` is the argument vector
  - array of strings with command-line arguments
- the int value is the return value
  - convention: return value of 0 means success, >0 means there was some kind of error
  - can also declare as `void` (no return value)

executing C programs (2).

- Name of executable followed by space-separated arguments
- `unix$ a.out 1 23 "third arg"`
- this is stored like this:



executing C programs (3).

- If no arguments, simplify:

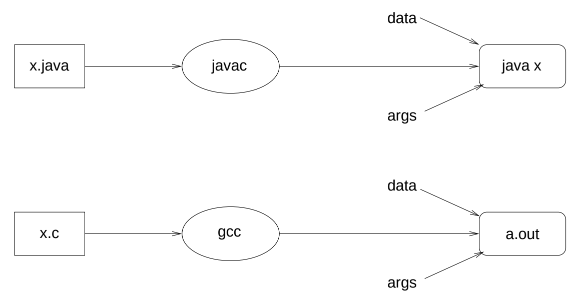
```
int main() {
  puts( "hello world" );
  exit( 0 );
}
```

- Uses `exit()` instead of `return()` — almost the same thing.

executing C programs (4).

- Java programs are compiled and interpreted:
  - `javac` converts `foo.java` into `foo.class`
  - class file is not machine-specific
  - *byte codes* are then interpreted by JVM
- C programs are compiled into object code and then linked into executables (to allow for multiple object files to work together):
  - `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

executing C programs (5).



compiling C programs (1).

- 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`

```
$ gcc hello.c
$ a.out
hello world!
```


compiling C programs (2).

- behavior of `gcc` is controlled by command-line switches:

-o *filename*	output file for object or executable
-Wall	display all warnings
-c	compiles but doesn't link
-g	insert code for debugger (gdb)
-p	insert code for profiler
-I	specify path for include files
-L	specify path for library files
-l	specify library
-E	pre-processor output only

compiling C programs (3).

- 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 math library:

```
gcc -o calc calc.c -lm
```


compiling C programs (4).

- 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 (1).

- 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:
  - define new macros
  - include files with C code (typically, "header" files containing definitions; file names end with `.h`)
  - conditionally compile parts of file
- gcc -E shows output of pre-processor
- can be used independently of compiler

C pre-processor (2).

```
#define name const-expression
#define name (param1,param2,...) expression
#undef symbol
```

- replaces name with constant or expression
- textual substitution
- symbolic names for global constants
- in-line functions (avoid function call overhead)
- type-independent code
- example: `#define MAXLEN 255`

C pre-processor (3).

- example:

```
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i)) { ...}
```

- becomes

```
if ((i) < 100)  {...}
```

- Caution: don't treat macros like function calls

```
#define valid(x) ((x) > 0 && (x) < 20)
```

is called like:

```
if (valid(x++)) {...}
```

and will become:

```
valid(x++) -> ((x++) > 0 && (x++) < 20)
```

and may not do what you intended...

C pre-processor (4).

- 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/sklar/programs/defs.h"
```


C pre-processor (5).

- conditional compilation
- pre-processor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- machine or OS-dependent code
- can be used to comment out chunks of code — bad!
  (but can be helpful for quick and dirty debugging :-)
- example:

```
#define OS linux
...
#if OS == linux
  puts( "good for you for running Linux!" );
#else
  puts( "why are you running something else???" );
#endif
```


C pre-processor (6).

- `ifdef`
- for boolean flags, easier:

```
#ifdef name
code segment 1
#else
code segment 2
#endif
```

- pre-processor checks if name has been defined, e.g.:

```
#define USEDB
```

- if so, use code segment 1, otherwise 2 now let's get down to actually writing some programs in C...

C comments.

- `/* any text until */`
- `// 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.

C data types (1).

- sizes and limits (may vary for machine; CUNIX is shown here):

type	**size in bytes** (on CUNIX)	**range**
char	8	$-128 \ldots 127$
short	16	$-32,768 \ldots 32,767$
int	32	$-2,147,483,648 \ldots 2,147,483,647$
long	32	$-2,147,483,648 \ldots 2,147,483,647$
float	32	$10^{-38} \ldots 3 * 10^{38}$
double	64	$2 * 10^{-308} \ldots 10^{308}$

- `float` has 6 bits of precision (on CUNIX)
- `double` has 15 bits of precision (on CUNIX)
- range differs from one machine to another
  - `int` is "native" size
  - e.g., 32 bits on 31-bit machines
  - there is always `short` and `long` and `int` will be the same size as one of these

C data types (2).

- you can also have *unsigned* values:

type	**size in bytes** (on CUNIX)	**range**
unsigned char	8	$0 \ldots 255$
unsigned short	16	$0 \ldots 65535$
unsigned int	32	$0 \ldots 4,294,967,295$
unsigned long	32	$0 \ldots 4,294,967,295$

- look at `/usr/include/limits.h`

C data type conversion (1).

```
#include <stdio.h>

void main( void ) {
  int i, j = 12;        /* i not initialized; j is */
  float f1, f2 = 1.2;  /* f1 not initialized; f2 is */

  i = (int)f2;          /* explicit: i <- 1, 0.2 lost */
  f1 = i;               /* implicit: f1 <- 1.0 */

  f1 = f2 + (float)j;  /* explicit: f1 <- 1.2 + 12.0 */
  f1 = f2 + j;          /* implicit: f1 <- 1.2 + 12.0 */
}
```


C data type conversion (2).

- implicit:

```
char b = '97';
int  a = 1;
int  s = a + b;
```

  - promotion: `char -> short -> int -> float -> double`
  - if one operand is `double`, the other is made `double`
  - else if either is `float`, the other is made `float`

```
int    a = 3;
float  x = 97.6;
double y = 145.987;
y = x * y;
x = x + a;
```


C data type conversion (3).

- explicit:
  - type casting

```
int    a = 3;
float  x = 97.6;
double y = 145.987;
y = (double)x * y;
x = x + (float)a;
```

- almost any conversion does something —
  but not necessarily what you intended!!

C data type conversion (4).

- example:

```
int x = 100000;
short s;
.
.
.
s = x;
printf("%d %d\n", x, s);
```

output is:

```
100000  -31072
```

“booleans” in C (1).

- C doesn’t have booleans
- emulate as `int` or `char`, with values 0 (false) and 1 or non-zero (true)
- allowed by flow control statements:

```
if ( n == 0 ) {
  printf( "something wrong" );
}
```

- assignment returns zero → false
- you can define your own boolean:

```
#define FALSE 0
#define TRUE 1
```


“booleans” in C (2).

- this works in general, *but beware:*

```
if ( n == TRUE ) {
  printf( "everything is a-okay" );
}
```

- if `n` is greater than zero, it will be non-zero, but may not be 1; so the above is NOT the same as:

```
if ( n ) {
  printf( "something is rotten in the state of denmark" );
}
```


the stdio library.

- Access stdio functions by
  - using `#include <stdio.h>` for prototypes
  - compiler links it automatically
- always defines `stdin, stdout, stderr`
- use for character, string and file I/O (later)

stdio functions: printf (1).

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

conversion character	**argument**	**description**
c	char	prints a single character
d or i	int	prints an integer
u	int	prints an unsigned int
o	int	prints an integer in octal
x or X	int	prints an integer in hexadecimal
e or E	float or double	print in scientific notation
f	float or double	print floating point value
g or G	float or double	same as e,E,f, or f — whichever uses fewest characters
s	char*	print a string
p	void*	print a pointer
%	none	print the % character

stdio functions: printf (2).

- some flags:

flag	description
-	left justify
+	print plus or minus sign
0	print leading zeros (instead of spaces)

- also specify field width and precision
- example:

```
printf( "i=%d s=%d f=6.3f m=43s",i,s,f,m );
```


stdio functions: scanf (1).

- `int scanf(const char *format, ...)` formatted output to stdout
- formatting:

conversion character	argument	description
c	char*	reads a single character
d	int*	reads a decimal integer
i	int*	reads an integer in decimal, octal (leading 0) or hex (leading 0x)
u	int*	reads an unsigned int
o	int*	reads an integer in octal
x or X	int*	reads an integer in hexadecimal
e, E, f, F, g or G	float or double	reads a floating point value
s	char*	reads a string
p	void**	reads a pointer

- more next Monday ... POINTERS!

stdio example.

```
#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;
}

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


data type conversion: integers to reals (1).

- example:

```
#include <stdio.h>

int main() {

  float f1 = 12.34;
  float f2 = 12.99;
  int   j, k;

  printf( "original values: f1=%f f2=%f\n",f1,f2 );
  j = (float)f1;
  k = f1;
  printf( "f1 ---> explicit j=%d, implicit k=%d\n",j,k );

  j = (float)f2;
  k = f2;
  printf( "f2 ---> explicit j=%d, implicit k=%d\n",j,k );

}
```

- output:

```
original values: f1=12.340000 f2=12.990000
f1 ---> explicit j=12, implicit k=12
f2 ---> explicit j=12, implicit k=12
```

data type conversion: integers to reals (2).

- example:

```
#include <stdio.h>
#include <math.h>

int main() {
  float f1 = 12.34;
  float f2 = 12.99;
  int   j, k, m, n;

  j = (float)f1;
  k = f1;
  m = ceil(f1);
  n = floor(f1);
  printf( "%f ---> explicit=%d, implicit=%d, ceil=%d, floor=%d\n",f1,j,k,m,n );

  j = (float)f2;
  k = f2;
  m = ceil(f2);
  n = floor(f2);
  printf( "%f ---> explicit=%d, implicit=%d, ceil=%d, floor=%d\n",f2,j,k,m,n );

}
```

- output:

```
12.340000 ---> explicit=12, implicit=12, ceil=13, floor=12
12.990000 ---> explicit=12, implicit=12, ceil=13, floor=12
```


using the math library (1).

- in the previous slide, the functions `ceil()` and `floor()` come from the C math library
- definitions:
  - `ceil( x )`: returns the smallest integer not less than `x`, as a `double`
  - `floor( x )`: returns the largest integer not greater than `x`, as a `double`
- in order to use these functions, you need to do two things:
  1. include the *prototypes* (i.e., function definitions) in the source code:

     `#include <math.h>`
  2. include the library (i.e., functions' object code) at link time:

     `unix$ gcc abcd.c -lm`
- exercise: can you write a program that *rounds* a floating point?

using the math library (2).

- some other functions from the math library (these are function *prototypes*):
  - `double sqrt( double x );`
  - `double pow( double x, double y );`
  - `double exp( double x );`
  - `double log( double x );`
  - `double sin( double x );`
  - `double cos( double x );`
- exercise: write a program that calls each of these functions
- questions:
  - can you make sense of `/usr/include/math.h`?
  - where are the definitions of the above functions?
  - what are other `math` library functions?

looping.

- 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!*
- `Cntrl-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?