today

- the MESS that is cs3157: update on ROOM and RECITATIONS
- HOMEWORK #1 WILL BE POSTED BY MIDNIGHT TONIGHT
- CHECK MODIFIED DATES ON SYLLABUS WEB PAGE
- compiling and the C preprocessor
- data types
- basic I/O (stdio)

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!
```

The C compiler gcc (2)

- Behavior 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	preprocessor output only

Using gcc

- 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` $\rightarrow$ `a.o`
  `gcc -c b.c` $\rightarrow$ `b.o`
  `gcc -o hello a.o b.o`
- Using math library:
  `gcc -o calc calc.c -lm`

Error reporting in gcc

- Multiple sources
- preprocessor: missing include files
- parser: syntax errors
- assembler: rare
- linker: missing libraries

Error reporting in gcc

- 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
- BUT don't ignore the warnings — compiler choice is often not what you had in mind
- does not flag common mistakes such as:
  `if (x = 0)` vs. `if (x == 0)`

gcc errors

- produces object code for each module
- assumes references to external names will be resolved later
- 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.
```

C preprocessor (1)

- the C preprocessor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it
- example:

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

- becomes

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

C preprocessor (2)

- preprocessor 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 preprocessor
- can be used independently of compiler

C preprocessor (3)

```
#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)
- mostly unnecessary for modern compilers
- type-independent code

C preprocessor (4)

- example: `#define MAXLEN 255`
- lots of system level `.h` files define macros
- invisible in debugger
- another example: `getchar()` and `putchar()` in `stdio` library
- 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 preprocessor (5)

- 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 preprocessor (6)

- conditional compilation
- preprocessor 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 preprocessor (7)

- `ifdef`
- for boolean flags, easier:

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

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

```
#define USEDB
```

- if so, use code segment 1, otherwise 2

C preprocessor (8)

- Advice:
- Limit use as much as possible
  - subtle errors
  - not visible in debugging
  - code hard to read
- much of it is historical baggage
- there are better alternatives for almost everything:
  - `#define INT16` $\rightarrow$ type definitions
  - `#define MAXLEN` $\rightarrow$ const
  - `#define max(a,b)` $\rightarrow$ regular functions
  - comment out code $\rightarrow$ CVS, functions
- limit to `.h` files, to isolate OS and machine-specific code

Now let’s get down to actually writing some programs in 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.

data types (1)

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

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

- `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

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`

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 */
}
```

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;
```

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!!

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 $\rightarrow$ 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 time ... POINTERS!

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!
```