CS 201

A Whirlwind Tour of C Programming

Gerson Robboy
Portland State University

A brute simple Makefile

```
all: sd test1 t1check test2

sd:  sd.c
    cc -g -o sd sd.c

test1:  test1.c
    cc -o test1 test1.c

test2:  test2.c
    cc -o test2 test2.c

t1check:  t1check.c
    cc -o t1check t1check.c

clean:
    rm sd test1 t1check test2 t1sort
```

Some things about Makefiles

Call it makefile or Makefile (big or little M)

- The “make” utility will use that by default

The first rule in the Makefile is used by default if you just say “make” with no arguments

The second line of each rule (the command) must start with a tab, not spaces!

A slightly more complex makefile

```
CC = gcc
CFLAGS = -Wall -O2
LIBS = -lm

OBJS = driver.o kernels.o fcyc.o clock.o

all: driver

driver: $(OBJS) config.h defs.h fcyc.h
    $(CC) $(CFLAGS) $(OBJS) $(LIBS) -o driver


driver.o: driver.c defs.h

kernels.o: kernels.c  ... h

fcyc.o: fcyc.c

clock.o: clock.c
```

A brute simple shell script

```
test1
 sd t1file t1sort
 t1check
 echo
 test2
 sd t2file t1sort
 t1check
 echo
```

- A plain text file containing commands
  - Known as a batch file in some O. S.'s
- Make sure the file has "executable" permission
- Invoke it by name, just like a binary program

argc and argv

main has two arguments:

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

- argc tells how many command line arguments there are, including the command itself
- argv is a pointer to an array of pointers to characters
- Example: find . –print
  - argc = 3
  - argv[0] = “find”
  - argv[1] = “.”
  - argv[2] = “-print”

The C preprocessor

```
bass> gcc -O2 -v -o p m.c a.c
cpp [args] m.c /tmp/cca07630.i
cc1 /tmp/cca07630.i m.c -O2 [args] -o /tmp/cca07630.s
as [args] -o /tmp/cca076301.o /tmp/cca07630.s
<similar process for a.c>
ld -o p [system obj files] /tmp/cca076301.o
   /tmp/cca076302.o
bass>
```

- cc or gcc, the compiler driver, invokes a bunch of things
  - The first one is cpp, the preprocessor
  - After that is cc1, the translator
- cpp converts the C source file to another C source file
  - expands #defines, #includes, etc.

Things you can tell the preprocessor

Included files:

```
#include <foo.h>
#include "bar.h"
```

Defined constants:

```
#define MAXVAL   40000000
```

Macros:

```
#define MIN(x,y)    ((x)<(y) ? (x):(y))
#define RIDX(i, j, n)   ((i) * (n) + (j))
```

Conditional compilation:

```
#ifdef …
#if defined( … )
#endif
```

What do you use conditional compilation for?

- **Debug print statements**
  - By defining or undefining one constant, you can include or exclude many scattered pieces of code
- **Code you think you may need again**
  - #ifdefs are more readable than commenting code out
- **Portability**
  - To multiple operating systems
  - To multiple processors
    - Compilers have “built in” constants defined
    - #if defined(__i386__) || defined(WIN32) || …
  - To different compilers for the same processor

Pointers

Pointers can point to any data type, including structures

```
pixel foo[32000];   // An array of pixels

// p is a pointer to a pixel, initialized to foo
pixel *p = foo;

&(foo[99]) is the same thing as (p+99)
```

- You can use subscripts with pointers
- Incrementing a pointer adds the increment times the size of the data type that it points to.

```
foo[99] is the same as *(p+99) or p[99]

foo[0].blue is the same thing as p->blue

foo[99].red is the same thing as (p+99)->red
```

Pointer example

Following a linked list

```
struct zilch listhead;
struct zilch *p = listhead;
   while (p != NULL) {
       ...  // Do stuff to *p
       p = p->next;
   }
```

What's the problem with this?

```
int zarray[34000];
char *zp = zarray;
int i;
for(i=0; i<34000; i++){
   *zp++ = i;
}
```

C pointer declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints