Practical C Issues:
Preprocessor Directives, Typedefs, Multi-file Development, and Makefiles

CS449 Fall 2016
Preprocessor Directives
#define

• Defines macros
  – Macro: rule that specifies textual replacement of one string for another
• Often used to assign names to constants

```c
#define PI 3.1415926535
#define MAX 10

float f = PI;
for(i=0; i<MAX; i++) ...
```
#define

• Good macros are generic (do not make assumptions about inputs)

• Good:
  – #define MAX(a,b) (a > b) ? a : b
  – Only assumes ‘a’ can be compared to ‘b’

• Not so good:
  – #define SWAP(a,b) {int t=a; a=b; b=t;}
  – Makes assumption that types are ‘int’

• Better
  – #define SWAP(T,a,b) {T t=a; a=b; b=t;}
#if

• #if <condition known to preprocessor>
  – Preprocessor emits code inside #if directive to the compiler only if condition is true
  – Condition evaluated at preprocessing time (cf. C if statement is evaluated at execution time)

• What does preprocessor know?
  – Values of #defined variables
  – Constants (0, 1, 2, “Linux”, “x86”, …)
  – Arithmetic (+, -, *, /, >, <, ==, &&, ||, …)
Example

```c
#include <stdio.h>

int main()
{
    #if 0
        printf("This is not compiled\n");
        I can doodle here when I am bored.
    #endif
    printf("This is compiled\n");
    return 0;
}
```
Example 2

```c
#include <stdio.h>
#define LIBRARY_VERSION 7
int main()
{
    #if LIBRARY_VERSION >= 5
        some_function_included_in_version_5();
        printf("This is compiled\n");
    #endif
    return 0;
}
```
#else

#if

...  

...  

#elif

...  

...  

#else

...  

...  

#endif
# if defined

- #if defined
  - Checks to see if a macro has been defined, but doesn’t care about the value
  - A defined macro might expand to nothing, but is still considered defined
Example

#include <stdio.h>
#define MACRO

int main()
{
    #if defined MACRO
        printf("This is printed\n");
    #endif
    printf("This is also printed\n");
    return 0;
}
#undef

- Undefines a macro:

```c
#include <stdio.h>
#define MACRO
#undef MACRO

int main()
{
    #if defined MACRO
        printf("This is not printed\n");
    #endif
    printf("This is printed\n");
    return 0;
}
```
Shortcuts

• `#if defined` → `#ifdef`
• `#if !defined` → `#ifndef`
Uses

• Handle code specific to a library, OS, processor, etc ...
• Turn on / off different features
  – Debugging:
    #ifdef DEBUG
    printf(…)
    #endif
  – More convenient debugging
    // easier to modify functionality of PrintDebug later
    #ifdef DEBUG
    #define PrintDebug(args...) fprintf(stderr, args)
    #else
    #define PrintDebug(args...) PrintDebug(args)
    #endif
Notes

• Can define variables from the commandline with \texttt{-D}
  \begin{itemize}
  \item \texttt{gcc -o test -DVERSION=5 test.c}
  \item \texttt{gcc -o test -DMACRO test.c}
  \end{itemize}
# Pre-Defined Macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE</strong></td>
<td>The currently compiled file</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>The current line number</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>The current date</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>The current time</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>Defined if compiler supports ANSI C</td>
</tr>
<tr>
<td>...</td>
<td>Many other compiler-specific flags</td>
</tr>
</tbody>
</table>
Other Preprocessor Details

- # - quotes a string
  - `#define CALL(f) { printf(#f); f(); }`
  - `CALL(foo) → { printf(“foo”); foo(); }`
- ## - concatenates two things
  - `#define CALL(f) f ## _debug ()`
  - `CALL(foo) → foo_debug()`
- #pragma: Change behavior of compiler
- #warning: Emit warning message
- #error: Emit error message and exit
Pragma Example

```c
#include <stdio.h>

#pragma message "Compiling " __FILE__ " using " __VERSION__
int main() {
    return 0;
}
```

```
>> gcc ./pragma.c
./pragma.c:3: note: #pragma message: Compiling ./pragma.c using 4.4.7
20120313 (Red Hat 4.4.7-4)
```

- Pragma message prints a message during compilation of file
- Use of two pre-defined macros: __FILE__ and __VERSION__
- Many more pragmas
  - To control compiler optimizations
  - To control code generation
#include <stdio.h>

#ifndef __i386__
#error "Needs i386 architecture."
#endif

int main() {
    return 0;
}

• Tests whether hardware platform is i386 (x86) and displays error
• Initially fails because default compilation target is x86_64
• ‘-m32’ option changes target to x86, allowing compilation to proceed
• #error: prevents compilation
#warning: allows compilation but with warning message
#include <stdio.h>
#define CALL(f) f ## _debug ()

void foo() {
    printf("foo normal\n");
}

void foo_debug () {
    printf("foo debug\n");
}

int main() {
    CALL(foo);
    return 0;
}
Multi-file Development
Multi-file Development

• Multi-file development breaks up a program into multiple files. Pros:
  – Parallel development involving multiple authors
  – Quicker compilation (only compile modified file)
  – Modularity (can reuse object file / library)
  – Encapsulation (easier to read / maintain)

• Use smallest scope to enforce encapsulation
  – Avoids polluting global namespace
    (Minimizes chances of name conflicts)
  – Minimizes scope of code to read to understand all uses of a function or variable
Local Scope

- **Scope:** Local (e.g. within a function)
- **Lifetime:** Automatic (duration of function)

```c
void f(...) {
    int x;
    ... 
    ...
}
```
Static Local Scope

- **Scope:** Local (e.g. within a function)
- **Lifetime:** Static (life of program)

```c
void f(...) {
    static int x;
    ...
}
```
Static Global Scope

- Scope: **File**
- Lifetime: **Static** (life of program)

```c
static int x;
void f(...) {
  ...
}
```
Global Scope

- **Scope**: Program
- **Lifetime**: Static (life of program)
- `extern` maybe be used to import variables from other files

File A

```c
int x;
```

File B

```c
extern int x;
```

Will refer to the same memory location
Example

```c
#include <stdio.h>

extern int x;
int f(int);

int main()
{
    x = 5;
    printf("%d", f(0));
    return 0;
}
```
Compiling

gcc a.c b.c

./a.out

5
Static

a.c

```c
static int x = 0;

static int f(int y)
{
    return x+y;
}
```

b.c

```c
#include <stdio.h>

extern int x;
int f(int);

int main()
{
    x = 5;
    printf("%d", f(0));
    return 0;
}
```
Compiling

gcc a.c b.c

/tmp/ccccyYUCUA.o(.text+0x6): In function `main':
  : undefined reference to `x'
/tmp/ccccyYUCUA.o(.text+0x19): In function `main':
  : undefined reference to `f'
collect2: ld returned 1 exit status
Header Files

- **Declarations** that need to be shared across multiple C (.c) files are put into **header** (.h) files
  - Functions (prototype declarations)
  - Variables (extern declarations)
  - `#defines` (macro declarations)
  - Type definitions
  - Other header files

- **Definitions** of symbols should be left to **C files**
  - Otherwise can lead to multiple definition link errors
Headers and Implementation

**mymalloc.h**

```c
void *my_buddy_malloc(int size);

void my_free(void *ptr);
```

**mymalloc.c**

```c
static void *base;

void *my_buddy_malloc(int size)
{
    ...}

void my_free(void *ptr)
{
    ...}
```
#include

- Copies the contents of specified file into current file
- `<>` means: look in a known location for includes
  - Usually `/usr/include`
- `""` means: look in the current directory or specified path (using `-I` option)
  - E.g. `gcc -I ~/local/include main.c`

```
#include <stdio.h>
#include "myheader.h"
```
- Looks for `stdio.h` under `/usr/include`
- Looks for `myheader.h` under cur. directory AND `~/local/include`
Including a Header File

• Driver program:
  ```c
  #include "mymalloc.h"
  ```

• Can now use those functions

• Compile:
  ```bash
  gcc -o malloctest mymalloc.c mallocdriver.c
  ```

• Why not also compile mymalloc.h?
  – Does not define or reference any symbols
    • Nothing to generate an object file out of
    • Hence nothing to link / compile
  – Only contains declarations to help compile .c files
Including a Header File Once

• Including same header twice can lead to compile errors
  – Redefinition of the same type, etc..
  – Sometimes not so obvious with multiple levels of nested headers

```c
#ifndef _MYHEADER_H_
#define _MYHEADER_H_
...
#endif
```

...Declarations only to be included once

```c
#endif
```
Makefiles

• Used with the GNU Make utility to build projects containing multiple files
• Goal: if any source files are modified, build smallest set required
  – By expressing what files depend upon others
• Composed of a collection of rules which look like
  \textit{target: dependencies action}
• Action must be followed by \texttt{<tab>}, not spaces
Makefile

malloctest: mymalloc.o mallocdriver.o
    gcc -o malloctest mymalloc.o mallocdriver.o

mymalloc.o: mymalloc.c mymalloc.h
    gcc -c mymalloc.c

mallocdriver.o: mallocdriver.c mymalloc.h
    gcc -c mallocdriver.c

clean:
    rm -f *.o malloctest
Dependency Graph

malloctest

mymalloc.o
mymalloc.c
mymalloc.h

mallocdriver.o
mallocdriver.c
Using a Makefile

• Build from scratch

thoth $ ls
Makefile mallocdrv.c mymalloc.c mymalloc.h
thoth $ make
gcc -c mymalloc.c
gcc -c mallocdrv.c
gcc -o malloctest mymalloc.o mallocdrv.o
thoth $ make
make: `malloctest' is up to date.

• Partial build after modifying mymalloc.c

thoth $ touch mymalloc.c
thoth $ make
gcc -c mymalloc.c
gcc -o malloctest mymalloc.o mallocdrv.o
Defining Variables in Makefiles

• Works like macros (text replacement)
• Syntax: `<name>` := ... or `<name>` = ...
• Example:
  – Instead of:
    malloctest: mymalloc.o mallocdriver.o
    gcc -o malloctest mymalloc.o mallocdriver.o
  – Can do:
    OBJECTS = mymalloc.o mallocdriver.o
    malloctest: $(OBJECTS)
    gcc -o malloctest $(OBJECTS)
Automatic Variables

• $@: The file name of the target. E.g.:
  malloctest: $(OBJECTS)
  gcc -o $@ $(OBJECTS)

• $<: The name of the first prerequisite. E.g.:
  mymalloc.o: mymalloc.c mymalloc.h
  gcc -c $<

• $^: The names of all prerequisites. E.g.:
  malloctest: $(OBJECTS)
  gcc -o $@ $^
Pattern Matching

• Character ‘%’ can stand for a pattern
• Example:

  % .o: % .c

  gcc –c $< -o $@

• What it means:
  – For all targets matching <some string>.o
  – Dependency is <that string>.c
  – Action is gcc –c <that string>.c –o <that string>.o

• Rule is used to produce any .o file from .c file
Concise Makefile

malloctest: mymalloc.o mallocdriver.o
        gcc -o $@ $^

%.o: %.c
        gcc -c $< -o $@

mymalloc.o: mymalloc.h
mallocdriver.o: mymalloc.h

clean:
        rm -f *.o malloctest
Make Utility Options

• Usage:
  make [-f makefile] [options] [targets]
• -f makefile: Can specify a different makefile
• targets: Can specify targets you want to build
• Options:
  – <name> = <value>: Define a variable.
  – -C <dir>: Change to directory dir before building.
  – -n: Dry run. Just print commands and don’t execute.
  – -d: Debug mode. Print verbose information.
Device Driver Makefile

obj-m := hello_dev.o

KDIR  := /u/SysLab/shared/linux-2.6.23.1
PWD   := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

• Default target of ‘make’ is first target (default: )
• -C option changes to kernel directory before building
• Invokes Makefile with variable ‘M’ defined as ‘PWD’ to build target ‘modules:’
Typedefs
typedef

typedef type-declaration synonym;

Examples:

typedef int * int_pointer;
typedef int * int_array;
Typedefs for Type Clarity

```c
void takes_int(int_pointer x) {
    *x = 3;
}

void takes_array(int_array x, int n) {
    int i;
    for(i=0; i<n; i++)
        printf("%d\n", x[i]);
}
```
Typedefs for Structures

With Typedef

typedef struct node {
  int i;
  struct node *next;
} Node;

Node *head;

Without Typedef

struct Node {
  int i;
  struct Node *next;
};

struct Node *head;

• Saves the trouble of typing ‘struct’ all the time
# Typedefs for Function Pointers

## With Typedef

```c
#include <stdio.h>
#include <stdlib.h>

typedef void (*FP)(int, int);

void f(int a, int b) {
    printf("%d\n", a+b);
}

g(int a, int b) {
    printf("%d\n", a*b);
}

int main() {
    FP fp1 = f;
    FP fp2 = g;

    fp1(2,3);
    fp2(2,3);
    return 0;
}
```

## Without Typedef

```c
#include <stdio.h>
#include <stdlib.h>

void f(int a, int b) {
    printf("%d\n", a+b);
}

g(int a, int b) {
    printf("%d\n", a*b);
}

int main() {
    void (*fp1)(int, int) = f;
    void (*fp2)(int, int) = g;

    fp1(2,3);
    fp2(2,3);
    return 0;
}
```
Function Pointers As Parameters

- In `<stdlib.h>`,

```c
void qsort (
    void *base ,
    size_t num ,
    size_t size ,
    compar_fn_t comparator
);

typedef int (*compar_fn_t) (const void *,
    const void *);
```
Function Pointers As Parameters

```c
int compare_ints(const void *a, const void *b)
{
    int *x = (int *)a;
    int *y = (int *)b;
    return *x - *y;
}

int main()
{
    int a[100];
    qsort(a, 100, sizeof(int), compare_ints);
}
```

- Function passed as parameter is called a *callback* function
- Device driver ‘read’ function was a callback function
Thank you

Please fill out the OMET Teaching Survey