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

CS449 Fall 2015
Preprocessor Directives
# define

- **Textual** Symbol Replacements

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

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

• Textual replacements with parameters:

• Good:
  – #define MAX(a, b) (a > b) ? a : b

• Not so good:
  – #define SWAP(a,b) {int t=a; a=b; b=t;}
#if

- #if <condition that can be evaluated by the preprocessor>

- What does preprocessor know?
  - Values of #defined variables
  - Constants
Example

```c
#include <stdio.h>

int main()
{
    #if 0
        printf("this is not printed\n");
    #endif
    printf("This is printed\n");
    return 0;
}
```
Example 2

```c
#include <stdio.h>
#define VERSION 5

int main()
{
    #if VERSION < 5
        printf("this is not printed\n");
    #endif
    printf("This is printed\n");
    return 0;
}
```
#else

#if

...

#elseif

...

...

#else

...

#endif
Example

```c
#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
”);
    #endif
    printf(“This is printed
”);
    return 0;
}
```
Shortcuts

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

• Handle System Architecture specific code
• Build program with different features
  – Debugging:
    ```c
    #ifdef DEBUG
    printf(…)
    #endif
    ```
  • Better debugging
    ```c
    #ifdef DEBUG
    #define PrintDebug(args…) fprintf(stderr, args)
    #else
    #define PrintDebug(args…)
    #endif
    ```
Notes

• Can define variables from the commandline with \texttt{\textendash D}
  
  \texttt{gcc -o test -DVERSION=5 test.c}
  \texttt{gcc -o test -DMACRO test.c}
## 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

• `##` - concatenates two things

• `#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;
}
```

- Pragma message prints a message during compilation of file
- Not use of two pre-defined macros: `__FILE__` and `__VERSION__`
- Many more pragmas
  - To control compiler optimizations
  - To control code generation
  - To convey program semantics

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

```c
#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
- Example of preprocessor usage for architecture specific code
# Concatenation example

```
#include <stdio.h>

#define PRINT(type, x) print_##type(x)

void print_int(int x) {
    printf("%d\n", x);
}

void print_char(char x) {
    printf("%c\n", x);
}

int main() {
    PRINT(int, 5);
    PRINT(char, 'H');
    return 0;
}
```

```
>> gcc ./concat.c
>> ./a.out
5
H
```

- ## concatenates two tokens into one token
- Useful when generating long identifiers with multiple components that can be given as arguments
Typedefs
typedef

typedef type-declaration synonym;

Examples:

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

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

<table>
<thead>
<tr>
<th>Typedef</th>
<th>Struct with Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>typedef struct node {</code></td>
<td><code>struct node {</code></td>
</tr>
<tr>
<td><code>   int i;</code></td>
<td><code>   int i;</code></td>
</tr>
<tr>
<td><code>   struct node *next;</code></td>
<td><code>   struct node *next;</code></td>
</tr>
<tr>
<td>} Node;</td>
<td>} Node;</td>
</tr>
<tr>
<td><code>Node *head;</code></td>
<td></td>
</tr>
</tbody>
</table>
Typedefs for Function Pointers

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

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

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

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

int main() {
    FP ar1 = f;
    FP ar2 = g;

    ar1(2,3);
    ar2(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);
}
```

- A function passed as a parameter is also called a *callback* function
Multi-file Development
Multi-file Development

- Multi-file development breaks up a program into multiple files
  - Multiple authors
  - Quicker compilation
  - Modularity
  - Encapsulation

- Use scoping to enforce encapsulation
  - Avoids polluting global namespace
  - Makes programs easier to maintain
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
```c
int x = 0;

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

#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
```c
static int x = 0;

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

```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/cccyUCUA.o(.text+0x6): In function `main':
    : undefined reference to `x'
/tmp/cccyUCUA.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 files are put into header files
  - Type declarations and typedefs
  - \#defines (macro declarations)
  - Functions (prototype declarations)
  - Variables (extern declarations)
  - Other header files

- Usually paired with an implementation file that has the definitions
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 the specified file into the current file
- <> means: look in a known location for includes
- " " means: look in the current directory or specified path (using –I option)
  - E.g. gcc –I ~/local/include main.c

```c
#include <stdio.h>
#include "myheader.h"
```
Driver

• Driver program:
  
  \#include "mymalloc.h"

• Can now use those functions

• Compile:
  
gcc -o malloctest mymalloc.c mallocdriver.c
Including a Header File Once

```c
#ifndef _MYHEADER_H_
declare _MYHEADER_H_

...Definitions of header to only be included once

#elseif
```

Makefiles

• Used with the GNU Make utility to build projects containing multiple files
• Goal: if any source files are modified, build smallest set required
• Express what files depend upon others
• Composed of a collection of rules which look like
  \textit{target: dependencies}
  \textit{action}
• Action must be followed by <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
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 $@
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 ‘default:’
- -C option uses specified directory as root of Makefile
- Invokes Makefile with variable ‘M’ defined as ‘PWD’ to build target ‘modules:’
Problem 1

- Write a macro that returns TRUE if its parameter is divisible by 10
Problem 2

• Write a macro `is_digit` that returns TRUE if its argument is a decimal digit
Problem 3

• Write a second macro `is_hex` that returns TRUE if its argument is a hex digit (0-9, A-F, a-f). The second macro should reference the first.
Problem 4

• Write a preprocessor macro that swaps two integers.