The Preprocessor

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 9

Preprocessor

- Commands to the compiler
- Include files, shortcuts, conditional compilation
- Command must start at beginning of line

Common preprocessor commands

- `#include`
- `#define`
- `#ifdef` / `#ifndef`

`#include`: Header Files

- Includes files: Literally copy-paste
- Typically header files

Header File

Declares

- External functions
- Variable types
- External global variables

Typically named `*.h` (or sometimes `*.hpp` for C++)

#include: Header Files

mylib.h

```
int max(int a, int b);
```

mylib.c

```
#include "mylib.h"
int max(int a, int b) {
    return (a > b ? a : b);
}
```

#include: Header Files

project.c

```
#include "mylib.h"

void foo() {
    ...
    m = max(p, q);
    ...
}
```

```
gcc -o project project.c mylib.c
```

Running just the preprocessor

To see the resulting code after preprocessing:

```
gcc -E -o preprocessed.c project.c
```

#define: Macros

Blind substitution inside file

```
#define     malloc      mymalloc
#define     maxsize     100

p = malloc(maxsize);
printf("Allocated %d bytes", maxsize);
```

is exactly the same as

```
p = mymalloc(100);
printf("Allocated %d bytes", 100);
```

#ifdef: Conditional compilation

project.c

```
#ifdef  DEBUG
#include "mylib.h"
#define  malloc      mymalloc
#define  free        myfree
#endif

...
p = malloc(100);
```

For debugging: `gcc -DDEBUG -o project project.c mylib.c`
For release: `gcc -o project project.c mylib.c`

#ifdef: Conditional compilation

mylib.h

```
void *mymalloc(int size);
void myfree(void *ptr);
```

#ifdef: Conditional compilation

mylib.c

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

void *mymalloc(int size) {
    void *ret = malloc(size);
    fprintf(stderr, "Allocating: %d at %p\n", size, ret);
    return ret;
}

void myfree(void *ptr) {
    fprintf(stderr, "Freeing: %p\n", ptr);
    free(ptr);
}
```

#include: Problems

mylib1.h

```
#include "mylib2.h"
```

mylib2.h

```
#include "mylib1.h"
```

#include: Solution

mylib1.h

```
#ifndef __MYLIB1_H
#define __MYLIB1_H
#include "mylib2.h"
#endif
```

mylib2.h

```
#ifndef __MYLIB2_H
#define __MYLIB2_H
#include "mylib1.h"
#endif
```

#define: More useage

- Use #define to create "inline functions"
  - Look like functions, but are not.
  - Expanded by prerpocessor into code.
  - Makes code more readable
  - More efficient than actual function call
    - No function call stack frame

```
#define INTS(n)   (int *) malloc(n * sizeof(int))

int *arr = INTS(5)
⇓
int *arr = (int *) malloc(5 * sizeof(int))
```

#define: More useage

- **CAUTION:** Macros get expanded "as-is". Might cause syntax issues or unwanted program bugs.

```
#define prod(a,b)    prod2(a, b * 10)
```

`prod(5,6)` ⇒ `prod2(5, 6 * 10)`

`prod(5,6+7)` ⇒ `prod2(5, 6+7 * 10)` BUG!!

#define: Solution

```
#define prod(a,b)     (prod2((a),(b)*10))
```

prod(5,6+7) ⇒ (prod2((5),(6+7)*10))

#define: More useage

```
#define oldfunc(a,b)    newfunc1(a); newfunc2(b);
```

```
oldfunc(5,6) ⇒ newfunc1(5); newfunc2(6)
```

```
for(i=0;i<5;i++) oldfunc(5,6);
⇒ for(i=0;i<5;i++) newfunc1(5); newfunc2(6);
```

BUG!!

#define: Solution

```
#define oldfunc(a,b)    do { \
    newfunc1((a)); newfunc((b)); \
} while (0)

for(i=0;i<5;i++) oldfunc(5,6);
⇒ for(i=0;i<5;i++) do {
  newfunc1(5); newfunc2(6);
} while(0);
```

#define: More problems

```
#define max(a,b)   ((a) > (b) ? (a) : (b))
```

```
max(p,q) ⇒ ((p) > (q) ?  (p) :  (q))
```

```
max(f1(),f2())
⇒ ((f1()) > (f2()) ?  (f1()) :  (f2()))
```
BUG!!

Solution: Be extra careful when calling a function inside code that could be a `#define`. Always use uppercase for macros to serve as reminder.

Serialization and Bit Operations

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Renato Paes Leme)

Fall 2011, Lecture 10

Serialization

- Sending data between programs
  - Disk
  - Network
  - Pipes
- Between programs on multiple hosts
  - Different endianness
  - Different architectures

Binary vs. Text

Binary...

- Compact
- Easy to encode/decode
- Faster

e.g. IP, TCP, AIM, ...

Text...

- Easily debugged
- (Can be) self-documenting
- Arch/Endian independent

e.g. HTTP, SMTP, MSN

Ok, but how?

What serialization solution to use?

- tpl library
- c11n library
- Google protocol buffers
- Customized solution

Which standard to use?

- XML, XDR, protocol buffer, ...
- Network protocol standards

Handling Endianness

Decimal: 3735928559
Binary: 11011110101011011011111011101111
Hex: 0xdeadbeef
Big Endian: 0xde 0xad 0xbe 0xef
Little Endian: 0xef 0xbe 0xad 0xde

Always in big-endian form when loaded into the CPU

Bit-Operations

AND-Mask (clear bits)

a & b

11011110101011011011110111101111	0xdeadbeef
&	&
00000000000000001111111100000000	0x0000FF00
=	=
00000000000000001011111000000000	0x0000be00

Bit-Operations

OR-Mask (sets bits)

a | b

```
11011110101011011011110111101111        0xdeadbeef
                    |                       |
00000000000000000101010100000000        0x00005500
                    =                       =
11011110101011011111111111101111        0xdeadFFef
```

Bit-Operations

Left-Shift

a << b

11011110101011011011111011101111	0xdeadbeef
<<	<<
8	8
=	=
10101101101111011101111**00000000**	0xadbeef**00**

Bit-Operations

Right-Shift

a >> b

11011110101011011011111011101111	0xdeadbeef
>>	>>
8	8
=	=
00000000110111101010110110111110	0x00deadbe[1]

[1] for unsigned ints only. For signed ints, the instead of zero-padding, the top-most bit is repeated

Bit-Operations

Compliment (flips bits)

$\sim$a

$\sim$11011110101011011011111011101111 = **00100001010100100100000100010000**

$\sim$0xdeadbeef = **0x21524110**

2's compliment representation for negative numbers:

`-x = ~x + 1`

Serialization

- Use structures for data-types
- Copy data in one-go
  `memcpy(dst, src, numbytes)`
- Use standard (big) endianness for multi-byte variables
- NEVER serialize pointer values. Why?

Threads

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 11

Processes vs. Threads

Processes . . .

- Multiple simultaneous programs
- Independent memory space
- Independent open file-descriptors

Threads . . .

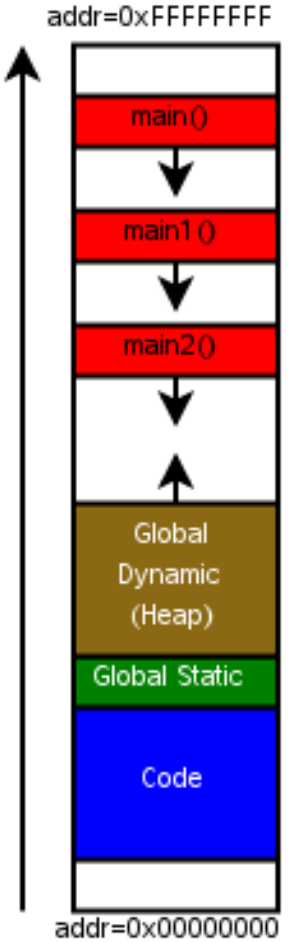
- Multiple simultaneous functions
- Share the same memory
- Share the same open file-descriptors

Threads Examples

- Graphical User Interfaces (GUIs)
  - The GUI is usually put on a separate thread from the "app engine"
  - GUI remains responsive even if app blocks for processing
- Web Browser Tabs
  - Each tab is managed by a separate thread for rendering
  - Web pages render "simultaneously"
  - Note: Google Chrome actually uses a separate process per tab

Threads

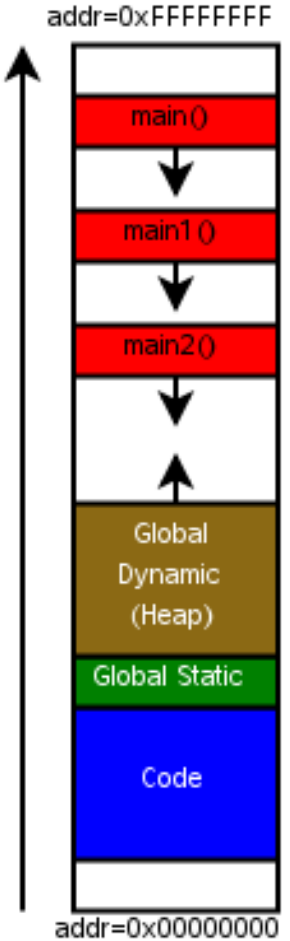
- One copy of the heap
- One copy of the code
- **Multiple** stacks



Threads

```
#include <pthread.h>
void *main2(void *arg) {
    ...
}
void *main1(void *arg) {
    ...
}
int main() {
    pthread_t id1, id2;
    pthread_create(&id1, NULL, main1, NULL);
    pthread_create(&id2, NULL, main2, NULL);
    ...
}
```

...think multiple processors (or cores)



pthread

Starting a thread

```
#include <pthread.h>
...
pthread_t id;
err = pthread_create(&id, NULL, entry_func, arg);
```

Body of a thread

```
void *entry_func(void *arg) {
        ...
```

Exiting current thread

```
        ...
        pthread_exit((void *)return_value);
}
```

pthread

Co-operative Multi-Threading on Single Processor

```
#include <sched.h>
...
    sched_yield()
```

- Store stack pointer, internal state etc. for current thread
- Restore stack pointer, internal state etc. for another thread
- Resume executing other thread

From the caller's perspective, sched_yield() blocks until the other thread calls sched_yield(). Allows multiple threads to share the CPU cooperatively.

pthread

Non-Cooperative Multi-Threading

Thread library (pthread) pre-empts thread when it invokes an OS function.

- Almost transparent when writing code
- Whole new class of bugs: Concurrency bugs
  - Multiple threads accessing same object concurrently
  - Solution: Locks – only one thread can grab lock
  - More of this is CS414/415

Goto, Exceptions, and Assembly in C

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 12

Switch Statement

- N-way `if` (N $>$ 2), but equality check only
- Only integers
  - But then many things in C are glorified integers
  - Notably, enums

Switch Statement

```
enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
...
enum days day = ...;
switch (day) {
    case Sat:
        ...
        break;

    case Sun:
        ...
        break;

    case Mon:
        printf("Sounds like someone has a case of the Mondays.\n");
    case Wed:
    case Fri:
        ...
        break;

    default:
        ...

}
```

Goto

- Unstructured control flow
  - (unlike `if`, `switch`, `for` etc.)
- Evil
- Except when it's not
- Especially, when it is the cleanest

Goto

```
    ...
    goto foo;
    ...

foo:
    ...
```

Goto

Extremely useful for

- breaking out of deeply nested loops
- handling errors and exceptions
  - by writing code that cleans up resources in reverse order of allocation
  - and jumping to the correct position in the list if allocation fails at some point

Exceptions (kinda)

- To break out of a deep call stack quickly
- Think goto breaking out of deep loops, but applied to function calls
- setjmp and longjmp

Inline Assembly

- For when no C statement exists for the task
- Or when the compiler isn't generating the assembly you want

```
asm("...assembly code...");  // Basic form

asm("code" : output);    // Assembly -> C

asm("code" : ... : input);    // C -> C
```

For more info check out:
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

Course Recap

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 13

Hello World!

```
#include <stdio.h>

void print_greeting()
{
  printf("Hello World!\n");
}

int main(int argc, char **argv)
{
  print_greeting();
  return 0;
}
```

Command Line Arguments

- When an application launches, the operating system can pass it *command line arguments*
- Optional and not required
- `int main(int argc, char **argv)`
  - `argc` - arguments count
  - `argv` - array of arguments as strings
  - application name counted as an argument, so `argc` is at least 1
- `int main()` is also valid if you don't care about command line arguments

Data Types 1/3 (Primitives)

- `int` - integer (size is platform dependent)
- `int32_t` - 32-bit integer on all platforms
- `float` - floating point number
- `char` - character
- `int[10]` - array of 10 integers
- `char[10]` - array of 10 characters (a string)
- ...

Data Types 2/3 (structs)

```
struct person
{
 char[20] name;
 int age;
 char[256] address;
};
```

- `struct` types hold collections of elements
- `struct person` (both words together) is now a "data type"
- Declare variables as such:
  `struct person john_doe;`
  - the '.' operator is used to access `struct` members
    `john_doe.age`

Data Types 3/3 (Pointers)

- Pointers are variables whose contents are interpreted as the memory addresses of other variables

Data Types 3/3 (Pointers)

- Pointers are variables whose contents are interpreted as the memory addresses of other variables
- `int *value;`
  `struct person *john_doe;`
- Operators relating to pointers
  - `*` - *dereference*: follow pointer and read data value
  - `&` - *address of*: get address of a variable (usually to store in pointer)
  - `->` - *access element*: access elements of a struct pointer. Equivalent to `(* _).`

Memory

- Stack
  - memory allocated statically by compiler
  - memory released back to system automatically after function returns
- Heap
  - memory allocated dynamically by programmer at run-time
  - memory has to be released back to system (freed) manually by programmer
  - use `malloc( size )` and `free( pointer )` to allocate and free memory
    - ```
      int *ptr = (int *) malloc(sizeof(int));
      free(ptr);
      ```
      ```

Frequently Used Libraries

- `stdio.h` - provides `printf`, `scanf`, `fgets` and other input/output functions
- `stdlib.h` - provides `malloc` and `free` functions
- `string.h` - provides `strcmp`, `strcpy` and other string manipulation functions
- `stdint.h` - provides `int32_t`, `uint32_t`, `int64_t`, `uint64_t` and other fixed size integers

Remember to include the correct library when using something provided by it!

Debugging

- Debugging is extremely valuable to determine what is going wrong with your program
- GDB is an interactive command line debugger
  - `break <function name or line number>` - sets a break point at a function def. or a line #
  - `print <variable name or expression>` - prints the value of a variable or an expression on program variables
  - `run <command line arguments>` - starts running the program with the given command line arguments
  - `help` - find help on more commands
- Compile your code with the -g flag for `gcc` to be able to debug the program with `gdb`

Terminal Input/Output

- Output to screen
  - `printf("Hi %d", 5);` - print formatted text
  - `puts("Hello World!");` - print a string
- Input from keyboard
  - `gets(buffer_array);` - read a single line from `stdin` into the `buffer_array`
  - `fgets(buf, 128, stdin);` - read a single line or at most 128 characters from `stdin` into `buf`
  - `scanf("%s %d", buf, &i);` - read from `stdin` and "parse" input according to given format

Many more variants that work with any stream type (for example good for file I/O)

File I/O

Opening and closing files

```
int fd;                          // File Descriptor
fd = open("/path/to/file", O_RDWR | O_CREAT);
close(fd);
```

Reading and Writing

```
char buf[4096]; int len;
len = read(fd, buf, 4096)
len = write(fd, buf, 4096);
```

WARNING: Size passed is only a **suggestion**. May read/write fewer than requested number of bytes. Return value is number of bytes actually read/written. MUST retry if not fully read/written.

Network I/O

Opening and closing network sockets

```
int sock;                              // File Descriptor
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
close(sock);
```

Internet Addresses

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(0x7F000001);
addr.sin_port = htons(8080);
```

Fill the address info manually or get the info automatically with `getaddrinfo()`.
See `man getaddrinfo`

Bitwise Operations

- Manipulate individual bits in a variable
- Useful for many things, one of which is serialization
- Operators
  - `a & b` - bitwise AND
  - `a | b` - bitwise OR
  - `a ^ b` - bitwise XOR
  - `~a` - bitwise one's complement
  - `a << b` - bitwise shift left
  - `a >> b` - bitwise shift right

Threads

Starting a thread

```
#include <pthread.h>
...
pthread_t id;
err = pthread_create(&id, NULL, entry_func, arg);
```

Body of a thread

```
void *entry_func(void *arg) {
        ...
```

Exiting current thread

```
        ...
        pthread_exit((void *)return_value);
}
```

Resources

- Dave's programming in C tutorials:
  http://www.cs.cf.ac.uk/Dave/C/CE.html
- Linux manual pages

  `man pthread_create`
  `man stdlib.h`
  - also available online at http://linux.die.net/
- Search online!

  Plenty of resources and tutorials online