COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

UNIVERSITY OF DENVER

The C Preprocessor

- Processes C code before it is being passed to the C compiler
- Preprocessor interprets directives
- Directives start with a # (which should be the first character on a line)
- Output of the preprocessor is still C code
- You can ask `gcc` to only do preprocessing using the `-E` option
- You can preprocess any text, not just C code

The #include directive

The most common directive is `#include FILENAME`.

- Any `#include FILENAME statement` is replaced by the preprocessor with the contents of `FILENAME`
- Most often used for C header files (`.h`) which provide (library) interface declarations
- Technically, anything can be `#include`-ed
- `#include` and other preprocessor operations can result in syntactic errors that are hard to find for beginners!

Conditional Compilation

- You can use `#if CONDITION` text `#endif` to cause the preprocessor to discard all text in between if `CONDITION` is zero
- Remember, preprocessing happens *before* compilation or execution!
- You can **not** use C variables or functions as `CONDITION`s.
- `CONDITION` can be a simple constant ("1", "0") or a macro expanding to a constant.

Macros

A macro is a textual substitution applied by the C preprocessor.

- Macros are defined using the #define directive
- Macros can be undefined using the #undef directive
- You can check if a macro is defined using the #ifdef directive

Macros in Headers

The most common use of Macros is preventing headers files from being included more than once:

```
/* myheader.h */
#ifndef MYHEADER_H
#define MYHEADER_H
/* actual header content here */
#endif
```

Macros as Constants

- C does not have constants
- The `const` keyword does something else!
- C uses macros instead of constants.

Macros as Constants: Example

```
#define PI 3.1415
#define YES 1
#define NO 0
#define ERROR -1
#define MY_ERROR_MESSAGE "Oh no, equal to PI!"
int larger_than_pi(float f) {
  if (f > PI) return YES;
  if (f < PI) return NO;
  fprintf(stderr, MY_ERROR_MESSAGE);
  return ERROR;
}
```

Macros as Inline Functions

Macros can have arguments:

```
#define MAX(a,b) ((a) > (b)) ? (a) : (b)
#define MIN(a,b) ((a) < (b)) ? (a) : (b)
```

Looks like a function, but...

```
int f() {
  int a = 0;
  int b = 1;
  int c = MAX(a++,b++);
  printf("%d %d %d\n", a, b, c);
}
```

What is the output if `MAX` is a function? What is the output if `MAX` is a macro? Why?

Macros and control flow

Macros can be too much fun:

```
#define HELLO(a,b) if ((a) < (b)) printf("Hello!")
int f() {
  int a = 0;
  int b = 1;
  int c = 2;
  if (a > c)
    HELLO(a,b);
  else
    printf("Party!");
}
```

After expansion...

```
int f() {
  int a = 0;
  int b = 1;
  int c = 2;
  if (a > c)
    if ((a) < (b)) printf("Hello!");
  else
    printf("Party!");
}
```

Is that what we wanted?

Avoiding dangling `else` issues

```
#define HELLO(a,b) do { if ((a)<(b)) \
             printf("Hello!");} while(0)
int f() {
  int a = 0;
  int b = 1;
  int c = 2;
  if (a > c)
    HELLO(a,b);
  else
    printf("Party!");
}
```

After expansion...

Macros can be more fun:

```
int f() {
  int a = 0;
  int b = 1;
  int c = 2;
  if (a > c)
    do { if ((a) < (b)) printf("Hello!"); }
    while(0);
  else
    printf("Party!");  }
```

The ## Operator

```
int fp(int a, int b) { return a+b; }
int fm(int a, int b) { return a*b; }
#define APPLY(a,b,o) f##o(a,b)
#define RUN(a,b,c,d,o) APPLY(a,APPLY(b,APPLY(c,d,o),o),o)
int main(int argc, char**argv) {
  printf("%d %d",
         RUN(1,2,3,4,p),
         RUN(1,2,3,4,m));
  return 0;
}
```

Macros and Types

```
#define fp(a,b) ((a)+(b))
#define fm(a,b) ((a)*(b))
#define APPLY(a,b,o) f##o(a,b)
#define RUN(a,b,c,d,o) APPLY(a,APPLY(b,APPLY(c,d,o),o),o)
int main(int argc, char**argv) {
  printf("%d %f",
         RUN(1,2,3,4,p),
         RUN(1.1,2.2,3.3,4.4,m));
  return 0;   }
```

Modular Compilation

- C compilers always only process one preprocessed unit of C code at a time
- This even applies if you run

```
$ gcc foo.c bar.c
```

- Header files are used to inform the C compiler about functions and variables available from other compilation units.

Declarations and Definitions

- Declarations introduce a symbol
- Definitions give the full details
- A symbol can have any number of (identical) declarations but only a single definition
- All symbols should be declared before they can be used (otherwise, the compiler will generate warnings)
- Declarations are needed for mutually recursive functions

Declarations and Definitions

- “public” functions are declared in header files
- “public” global variables can be declared in header files
- `structs` and `unions` can be declared or defined in headers
- `structs` and `unions` must be defined before `sizeof` can be used
- Macros can **not** be declared

Examples for Declarations

```
float sin(float);
float sin(float f);
void run(void);
struct Foo;
union Bar;
extern int flag;
```

The `extern` keyword is mandatory for global variable declarations!

Declarations and Definitions

- Only declare what you must declare for compilation without warnings
- Only declare functions in headers that are part of the API that is supposed to be used by other C files
- Avoid declaring global variables
- W32 does NOT allow libraries to declare global variables!

`static`

- `static` limits the scope of a declaration or definition to the current compilation unit
- Use `static` on variables and functions as much as possible
- `static` on local variables has a different meaning!

Example for `static`

```
static int b;
static int m() {
  static int a;
  return b * a++;
}
int main(int argc, char** argv) {
  b = 4;
  printf("%d %d %d\n", m(), m(), m());
}
```

UNIVERSITY OF DENVER

Linking

- Linking is automatic unless `-c` is passed to `gcc`
- Linking maps uses of declared symbols to definitions in other compilation units
- Symbols that are declared `static` are NOT eligible for use by other compilation units or the linker
- If symbols were declared and used but are not defined anywhere, linking may fail!
- Symbols from external libraries (like GNU libc) will be resolved by the loader

Types of Binaries

- Static Libraries: resolved by linker
- Shared Libraries: resolved by loader
- Programs: contain `main`

For now, you will always use `gcc` to create programs. Creating libraries will be discussed in lecture 7.

Loading

- A loader loads a binary and (shared) libraries that the binary depends on into memory
- The loader then modifies the code to match unresolved symbols from the binary to the respective symbols of the libraries
- `ldd` shows which libraries the loader will load to resolve symbols

We will have more fun with the loader in lecture 10.

gdb **Invocation**

- $ gdb binary-name
- $ gdb binary-name core-file
- Make sure binary is compiled with option −g
- Using −O0 (no optimizations) might also be useful

Using gdb

- (gdb) run ARGS
- (gdb) attach PID
- (gdb) break FUNCTION
- (gdb) break FILENAME:LINE
- (gdb) bt DEPTH

Using gdb

- (gdb) continue
- (gdb) s[tep]
- (gdb) n[ext]

Using `gdb`

- (gdb) info args
- (gdb) info locals
- (gdb) info threads

Printing and eXamining

- (gdb) print EXPRESSION
- (gdb) print array-ptr@size
- (gdb) x[/format] address
- (gdb) x/s a $\equiv$ (gdb) print (char*) a
- (gdb) x/NNNi main

Variables

- `gdb` automatically creates a variable ($NN) for any examined expression
- You can define your own using `set $NAME = EXPRESSION`

Creating Functions

- (gdb) define NAME
- > while x > 50
- > step
- > end
- > print i
- > end

Arguments are $arg0, ..., $argN.

Executing Commands at Breakpoints

- (gdb) break filename.c:line
- (gdb) commands
- > silent
- > set x = 42
- > continue
- > end

Watchpoints

- (gdb) watch x – write only
- (gdb) rwatch x – read only
- (gdb) awatch x – read/write

Read watchpoints may only work with hardware support.

Remember

- The best way to eliminate bugs is to not write them
- The best debugger is your own brain
- Good testcases make debugging easier
- Not all bugs cause visible problems

Questions

?