Lecture 8

Dr. Jack Lange

Preprocessing

- Compiling C programs is a multistage operation
  - Before code is actually compiled it is "pre-processed"
- C Pre-Processor
  - Basically just a simple syntax for text manipulation
  - Produces the final text that is used by the compiler
- We've seen an example already
  - `#include <stdio.h>`
  - Replace the line with the raw contents of the specified file

Macros

- Macros lines with an '#' as the first character
  - More than just #include
- #define
  - 2nd most common macro
  - Automated "find-and-replace" operation at compile time
  - **#define <PATTERN> <REPLACEMENT>**
  - Like declaring a variable but only in text!
- Macros are almost like a separate programming language
  - Not compiled (only evaluated by the pre-processor)

#define Example

- Usually used for commonly used constants
  - E.g. buffers usually have a standard size
  - Allows changing the constant in one location
    - Don't need to track down every usage
    - Avoids a lot of bugs

```
#define BUF_SIZE 100

int main() {
    char buf[BUF_SIZE];
    memset(buf, 0, BUF_SIZE);
    fgets(buf, BUF_SIZE, stdin);
}
```

Conditional Macros

- Enable/Disable blocks of code before compilation

```
#ifdef MACRO
…
#endif
```

AND

```
#ifndef MACRO
…
#endif
```

- #ifdef
  - Useful for turning on/off debugging features
  - Useful for compile time configuration
- Example:

```
#ifdef ENABLE_DEBUGGING
   printf(lots of state values);
#endif

> gcc -DENABLE_DEBUGGING test.c -o test
```

Conditional Macros

- Enable/Disable blocks of code before compilation

```
#ifdef MACRO
…
#endif
```

AND

```
#ifndef MACRO
…
#endif
```

- #ifndef
  - Useful for header file composition
    - Include a header file only once (why is this important?)
- Example: header.h

```
#ifndef __HEADER_H__
#define __HEADER_H__

int global_var;

#endif
```

Function Macros

- Text replacement that understands arguments
  - Looks like a function definition
  - Only matches text that looks like a function : "()"

```
#define FOO(x) (x++)
#define MAX(x, y) (x > y) ? x : y;
```

- Only text replacement
  - The function body can be **ANY** text
  - Syntax is not checked until actual compilation
  - Can be ***extremely*** confusing and convoluted
    - **BEWARE COMPLEX MACROS**

Nifty printf

```
#define ERROR(fmt, args…) \
   printf("error> %s (%d): " fmt, __FILE__, __LINE__, ##args);


int foo() {
   …
   if (error) ERROR("Error Occurred");
   …
}
```

- Automatically adds the filename and line number to output string
  - Transforms ERROR() to an extended printf() call

Typecasting

- C provides basic type checking
  - If you try `int x = "foo";` the compiler will catch it
- However, you can easily override this
  - Example: **typecase1.c**

```
char * str = "foo";
int    x   = *(int *)str;
```

- The universal type : **void *** *(Example: **typecast2.c**)*
  - A raw memory address, can point to anything
  - Any pointer can be implicitly cast to a 'void *'

```
int foo(void * x);

void * ptr = NULL;
char * str = "hello";
int x = 4;

foo(ptr);
foo(str);   // This is OK
foo(&x);    // This is also OK
```

Function Pointers

- Every part of the program is stored in memory
  - Code, data, everything
  - C allows you direct access to memory
  - So we can access the code itself
  - *Or, we can execute data as code*
- **Function pointers:** *Functions as a data type*
  - We can assign a function to a variable
    - Just a memory address where the code is located
  - Then "call" the variable
    - Force the CPU to jump to that location in memory

Function Pointer Example

```
int bar(int x, int y) {return x + y);

int (*foo)(int x, int y);

foo = bar;
foo = &bar; // Same thing
foo(1, 2);
```

- Looks similar to regular pointers
  - But there is a subtle difference

```
int * x;     // x is a pointer to an int
int (*x)(); /* x is a pointer to a function
             *   that returns an int
             */
```

Using function pointers

- Allow object oriented behaviors
  - Can embed function pointers inside a struct
  - Starts to look like a basic object
    - Instance based methods
    - Local variables