The Preprocessor

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- The preprocessor
- Macros
- Macros with parameters
- Special parameter expansion
- Conditional compilation
- Include guards

Meet the Preprocessor

- The preprocessors : a dumb-ish text manipulation stage before compilation.
- What’s it good for?
  - Including fragments of source code (e.g., headers)
  - Defining constants (via macros)
  - Substituting for inline functions (via macros)
  - Conditional compilation

You Already Know

- The preprocessor is the first step in compiling your code
  - It does basic text-based operations on your source code
- You can use it to define named constants
  - During preprocessing, occurrences of this token get replaced by this definition.

```
#define SIZE 256
```

  - This is called a *preprocessor macro*
  - This replacement is called *macro expansion*

Macro Expansion

- The preprocessor is a little smart, it looks for whole identifier matching the macro name.

```
#define SIZE 1024
```

- It will substitute in situations like these:

```
int i = SIZE ;
```

```
f( str, (SIZE+1)*2);
```

- But not a situation like this:

```
int list[MAXSIZE];
```

Preprocessor Macros

- Guess what. Preprocessor macros can expand to multiple tokens.
- This can be useful.

```
#define FAIL exit( 1 )
```

- It can also be a source of some common mistakes:

```
#define SIZE 256;
```

```
int list[ SIZE ];
```

```
int list[ SIZE; ];
```

The preprocessor is dumb.
You have to provide the smarts.

Looking Inside

- It can be hard to detect these kinds of errors.

```
#define SIZE 256;
```

  - We don't get to see the intermediate code generated by the preprocessor.
  - But, we can … if we know how to ask.
- The –E option to gcc will send preprocessed output to standard out.

```
gcc -E myProgram.c
```

Get ready for a lot of output.

Preprocessor Directives

- Macro definitions are one example of a *preprocessor directive*
  - Starting with a #
  - Continuing to the end-of-line, but you can escape the newline with \

```
#define LONG_MACRO 1 + 2 + 3 + 4 + 5 + 6 + 7 \
+ 8 + 9 + 10
```

  - Handled by the preprocessor
  - Removed before the compiler gets the code.

Macros within Macros

- Macro definitions can contain other macros

This also works.

```
#define PI 3.14
#define TWO_PI 2 * PI
```

```
… TWO_PI …
```

```
… 2 * PI …
```

```
… 2 * 3.14 …
```

```
#define TWO_PI 2 * PI
#define PI 3.14
```

```
… TWO_PI …
```

```
… 2 * PI …
```

```
… 2 * 3.14 …
```

Macro Parameters

- Macros can take parameters

```
#define times2( x ) x * 2
```

Soon, we'll see that this is a bad idea.

Beware. No space here.

- When the macro expands, the parameter gets copied into the definition

```
int j = times2( i );
```

→

```
int y = i * 2;
```

```
a[i] = times2(a[k]) + 1;
```

→

```
a[i] = a[k] * 2 + 1;
```

- This is a mechanism for call-by-name

Macros as Functions

- Macros can substitute for inline-functions.

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

```
int i = …, j = …;
int k = MAX( i, j );
```

```
int i = …, j = …;
int k = i > j ? i : j;
```

No function-call overhead.

- And, we can use them to write code that’s independent of type.

```
double a = …, b = …;
double c = MAX( a, b );
```

```
double a = …, b = …;
double c = a > b ? a : b;
```

Planning for Expansion

- So far, all of our macros have been poor example ... like this one:

```
#define times2( x ) x * 2
```

- What happens if you use it like this?

```
int j = times2( i + 1 );
```

```
int j = i + 1 * 2;
```

Oops, I think we wanted (i + 1) * 2

- We can use parentheses to suppress this problem.

```
#define times2( x ) (x) * 2
```

```
int j = times2( i + 1 );
```

```
int j = (i + 1) * 2;
```

Planning for Expansion

- We can get similar problems with precedence of surrounding expression syntax.

```
#define times2( x ) (x) << 1
```

- What happens if you use it like this?

Oops, I think we wanted
(i << 1) + 1

```
int j = times2( i ) + 1;
```

```
int j = (i) << 1 + 1;
```

- Parentheses around the whole thing will fix this.

```
#define times2( x ) ((x) << 1)
```

```
int j = times2( i ) + 1;
```

```
int j = ((i) << 1 ) + 1;
```

Protecting the Macro Definition

- So, our MAX macro would really look more like:

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

- If a macro has multiple statements
  - we probably need to protect them within a block.

```
#define PRINT2( x, y ) { \
    printf( “%d\n”, x ); \
    printf( “%d\n”, y ); \
}
```

Macros vs Functions

- All of these examples show how macros can work like functions
- … but, they can be more difficult to use correctly.

- There’s still one more problem with using macros as functions …

Macros and Side Effects

- Notice, a parameter may get copied more than once after macro expansion

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

- Here, it's may hurt performance.

```
double z = MAX( sqrt(x), sin(y) );
```

```
double z = ((sqrt(x)) > (sin(y)) ? (sqrt(x)) : (sin(y)));
```

Macros and Side Effects

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

- But, here it could cause unexpected behavior.

```
int z = MAX( x++, y++ );
```

One of us will get incremented twice.

```
int z = ( (x++) > (y++) ? (x++) : (y++) );
```

- If x == 3 and y == 4, what will x, y and z be after this assignment?

Macros and Side Effects

- It's the programmer's job to consider possible errors like this.

```
int z = MAX( x, y );
x++, y++;
```

That's better.

- This is why the documentation warns you about calls that may be implemented as a macro
  - getc()
  - putc()
  - …

Macros Can See The Parameters

- Macros have access to the text of their parameters
  - This is an example of call-by-name
- With this, we can do things a function can't

```
#define SWAP(a, b, type) \
{ \
   type temp = a; \
   a = b; \
   b = temp; \
}
…
   int x = 5;
   int y = 7;
   …
   SWAP(x, y, int);
```

Macro Expansion of Macro Expansion of …

- Consider this handy little macro

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

- What if you need the maximum of 3 or more items?
- No problem for 3:

```
MAX( a, MAX( b, c) )
```

- Or for 4:

```
MAX( MAX( a, b ), MAX( c, d ) )
```

Macro Expansion of Macro Expansion of …

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

- But, things could get ugly.

```
MAX( a, MAX( b, c) )
```

↓

```
( (a) > (MAX( b, c)) ? (a) : (MAX( b, c)) )
```

↓

```
( (a) > (( (b) > (c) ? (b) : (c) ) )
    ? (a)
    : (( (b) > (c) ? (b) : (c) ) ) )
```

Controlling Expansion

Quoting with the # character

```
#define TMPFILE(dir,fname) #dir "/" #fname
…
char s[] = TMPFILE(/usr/tmp,test1) ;
```

```
char s[] = "/usr/tmp" "/" "test1" ;
```

Concatenation with the ## characters

```
#define CAT(x,y) x ## y
…
a = CAT(b,123) ;
```

```
a = b123;
```

Help with Debugging

- Here, call-by-name lets us do things we couldn't with a function.
- We can give more context in our debug output

```
// Macro to print the value of an int, along with
// its name.
#define REPORT_I( X ) printf( "%s = %d\n", #X, (X) )

…

  REPORT_I( i );
  REPORT_I( total );
```

`#include`

- Inserts into the source code the contents of another file
  - often called a *header* file (filetype: **`.h`**)

```
#include <stdio.h>
#include "mydefs.h"
```

standard library header file

user defined header file

Where does gcc look for these files?

- installation dependent for < > (but often **/usr/include**)
- same directory as source code file for " "
- other locations controlled by `gcc` **`–I`** option

`#include`

- Frequently part of header files:
  - constant definitions
  - Extern declarations
    - Global variables (marked extern)
    - Function prototypes
  - type definitions
- When the header file changes, all source files that **`#include`** it have to be recompiled
  - i.e., there is a dependency of this source code on the contents of the header file

Conditional Compilation

- To control what source code gets compiled
- Common uses
  - to resolve, at compile time, platform (machine- or OS-) dependencies
  - to compile (or not) debugging code
- Here's what conditional compilation looks like

```
#if...

someCodeYouMay();
orMayNotWant();

#endif
```

Conditional Compilation

- We have lots of (related) directives for writing condition compilation
  - **`#if / #ifdef / #ifndef`**
  - **`#elif / #else`**
  - **`#endif`**

Preprocessor Expressions

- We can ask the preprocessor if macros are defined.

```
#if defined(X)
```

- This is so common, it has its own syntax.

```
#ifdef X
```

- We can ask about particular values.

```
#if X == 25
```

- We can even build compound conditionals

```
#if X > 25 && Y < 30
```

Conditional Compilation: Example

```
#if defined(LINUX)
  #define HDR "linux.h"
#elif defined(WIN32)
  #define HDR "windows.h"
#else
  #define HDR "default.h"
#endif

#include HDR
```

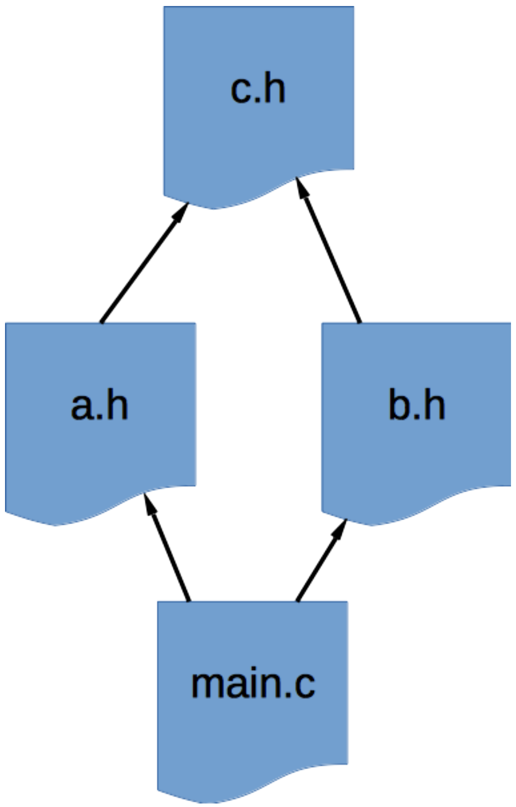
- And when compiling this program, can define what **SYSTEM** is by using the **-D** option to **gcc**

```
gcc -DWIN32 myprog.c …
```

```
#include "windows.h"
```

Include Guards

- It’s possible to include the same header more than once … even without trying to.
- Is this bad?
  - Well, that’s extra work for the compiler
  - And, if the header contains definitions, we’re in trouble.
  - You can declare something as many times as you like, but you better only define it once.
- How can we fix this?
  - You guessed it, with the preprocessor.



Include Guards

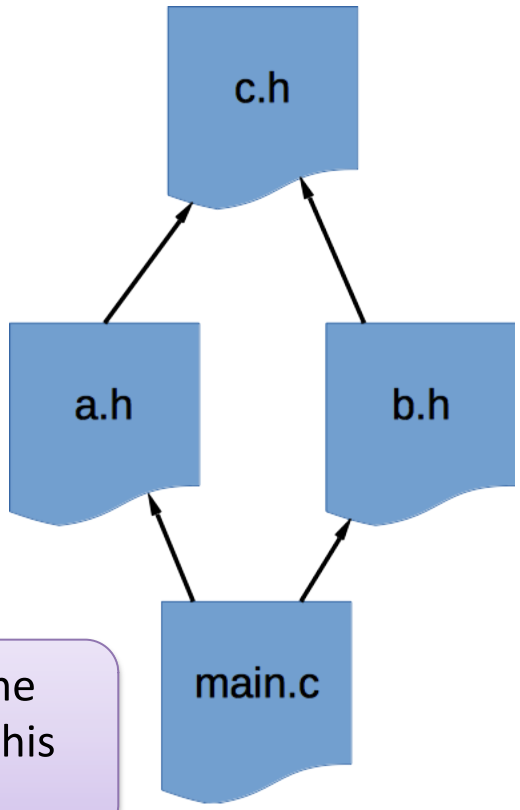
- We can use trick the preprocessor into discarding the contents of a header after it's already been processed once.

```
#ifndef C_H
#define C_H

…
… all the stuff inside c.h
…

#endif
```

Some name unique to this header.



Help with Debugging

- The preprocessor can help with debugging
- We can use it to exclude blocks from compilation.

```
#ifdef DEBUG

   …;
   #ifdef DEBUG
      …;
   #endif
   …;

#endif
```

Unlike comments, these will nest.

Help with Debugging

- Here, call-by-name lets us do things we couldn't with a function.
- We can give more context in our debug output
- Or, disable them completely with a recompile.

```
#if defined(DEBUG)
#define REPORT_I( X ) printf( "%s = %d\n", #X, (X) )
#else
#define REPORT_I( X )
#endif

…

  REPORT_I( i );
  REPORT_I( total );
```

Help with Debugging

- When we compile, we can choose which macros to enable.

```
gcc -DDEBUG -std=c99 -Wall …
```

Any macro you want to define.

I'm defined to **1**

```
gcc -DNAME=BILL "-DMESSAGE=HELLO WORLD" -std=c99 …
```

I'm set to BILL

With help from the shell, you can include spaces.