

CS 2211

Systems Programming

Compiler Directives

The C Preprocessor

- The C preprocessor (**cpp**) changes your source code based on instructions, or preprocessor directives, embedded in the source code.
- The preprocessor creates a “new” version of your program and it is this new program that actually gets compiled.
 - Normally, you do not see these “new” versions on the hard disk, as they are deleted after compilation.
 - You can force the compiler to keep them to see the results.
- Each preprocessor directive appears in the source code preceded by a **#** sign.

The #define Directive

- Simple substitution Macros

```
#define text1 text2
```

- This tells the compiler to find all occurrences of “text1” in the source code and substitute “text2”.

- Usually used for constants:

```
#define MAX 1000
```

- Generally use upper case letters (by convention).
- Always separate by white space.
- No trailing semi-colon (think about it...)

- An example:

```
- #define PRINT printf
```

```
PRINT("hello, world");
```

Function Macros

- You can also define more complex macros:

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

.....

```
printf("%d", 2 * max(3+3, 7)); /* is equivalent to */
printf("%d", 2 * ( (3+3) > (7) ? (3+3) : (7) ) );
```

- The parentheses are important! For example:

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

```
printf("%d", 2 * max(3+3, 7)); /*is equivalent to */
printf("%d", 2 * 3+3 > 7 ? 3+3 : 7 );
```

Function Macros Should be Used with Care

```
#define max(x,y) ((x)>(y)?(x):(y))  
.....  
int n, i=4, j=3;  
  
n= max( i++, j );  
    /* Same as n= (( i++ )>( j )?( i++ ):( j )) */  
printf("%d,%d,%d", n, i, j);
```

- The output is:
 - 5, 6, 3
- If **max** was a function, the output would have been:
 - 4, 5, 3


Conditional Compilation

- The pre-processor directives `#if`, `#ifdef`, `#ifndef`, and `#endif` tell the compiler

Any Constant Expression

- non-zero is true
 - compile statement_block_1
- zero is false
 - don't compile statement_block_1

- Structure:



```
#if condition_1
    statement_block_1
#elif condition_2
    statement_block_2
...
#elif condition_n
    statement_block_n
#else
    default_statement_block
#endif
```

Conditional Compilation

- For the most part, the only things that can be tested are the things that can be defined by `#define` statements.
- An example:

```
#define ENGLAND 0
#define FRANCE 1
#define ITALY 0
#if ENGLAND
    #include "england.h"
#elif FRANCE
    #include "france.h"
#elif ITALY
    #include "italy.h"
#else
    #include "canada.h"
#endif
```

Conditional Compilation

- Conditional compilation can also be very useful for including “debugging code”
 - When you are debugging your code you probably print out some information during the running of your program.
 - However, you may not need want these extra print outs when you release your program. So, you need to go back through your code and delete them.

- Instead, you can use `#if` `#endif` to save you time:

```
#define DEBUG 1
.....
#if DEBUG
    printf("Debug reporting at function my_sort()!\n");
#endif
.....
```


Conditional Compilation

Usually people use a preprocessor function as the condition of compilation:

`defined (NAME)`

Returns true if `NAME` has been defined; else false

An example:

```
#define DEBUG
#if defined ( DEBUG )
    printf("debug report at function my_sort() \n");
#endif
```

Note: This only depends on if `DEBUG` has been defined. But has nothing to do with which value `DEBUG` is defined to.

Can also use the notation `#ifdef NAME` instead.

Conditional Compilation

The **#undef ...** directive makes sure that **defined(...)** evaluates to false.

An example:

Suppose at the first part of a source file, you want **DEBUG** to be defined.
At the last part of the file, however, you want **DEBUG** to be undefined...

A directive can also be set on the Unix command line at compile time:

cc -DDEBUG myprog.c

Compiles **myprog.c** with the symbol **DEBUG** defined as if

#define DEBUG

was in written at the top of **myprog.c**.

The #include Directive

- We've seen lots of these already.
- This directive causes all of the code in the included file to be inserted at the point in the text where `#include` appears.
- The included files can contain other `#include` directive.
 - Usually limited to 10 levels of nesting
- `< >` tell the compiler to look in the standard include directories first.
- `" "` tells the compiler to treat this as a Unix filename.
 - Relative to directory containing file if a relative pathname.
 - Relative to root with an absolute pathname.
 - But most compilers also search for the standard include directory if it cannot find the file at the specified path.

Inline Functions

- Recall the two different ways to compute the maximum number between two integers:
 - `#define max(a,b) ((a)>(b)? (a):(b))`
 - `int max(int a, int b) { return a>b?a:b; }`
- Function calls need to jump to another part of your program and jump back when done. This needs to:
 - Save current registers.
 - Allocate memory on the stack for the local variables in the function that is called.
 - Other overhead
- Therefore, the macro approach is often more efficient, since it does not have function call overhead.
 - But, this approach can be dangerous, as we saw earlier.

Inline Functions

- Modern C and C++ compilers provide “inline” functions to solve the problem:

- Put the inline keyword before the function header.

```
inline int max(int a, int b) {  
    return a>b?a:b;  
}
```

- You then use it as a normal function in your source code.

- `printf("%d", max(x, y));`

- When the compiler compiles your program, it will not compile it as a function. Rather, it just integrates the necessary code in the line that `max()` is called in to avoid an actual function call.

- The above `printf(...)` is compiled to be something like:

- `printf("%d", x>y?x:y);`

Inline Functions

- Modern C and C++ compilers provide “inline” functions to solve the problem:

- Put the inline keyword before the function header.

```
inline int max(int a, int b) {  
    return a>b?a:b;  
}
```

- You then use it as a normal function in your source code.

- `printf("%d", max(x, y));`

- When the compiler compiles your program, it will not compile it as a function. Rather, it just integrates the necessary code in the line that `max()` is called in to avoid an actual function call.

- The above `printf(...)` is compiled to be something like:

- `printf("%d", x>y?x:y);`

Inline Functions

- Writing the small but often-used functions as inline functions can improve the speed of your program.
- A small problem in doing so is that you have to include the inline function definition before you use it in a file.
 - For normal functions, only the function prototypes are needed.

Inline Functions

- Therefore, inline functions are often defined in header (.h) files.
 - Once you include the header file, you can use
 - Inline functions whose definitions are in that header file.
 - Normal functions whose prototypes are in that header file.
- Another small problem is that some debuggers get confused when handling inline functions
 - sometimes it is best to inline functions after debugging is finished.

C Compiler Directives

END OF Compiler Directives

