C PREPROCESSOR

PROGRAMMING LANGUAGES

BY

Z. CIHAN TAYSI

OUTLINE

- Macro processing
  - Macro substitution
  - Removing a macro definition
  - Macros vs. functions
  - Built-in macros
- Conditional compilation
  - Testing macro existence
- Include facility
- Line control

MACROS

- All preprocessor directives begin with a pound sign (#), which must be the first nonspace character on the line
- Unlike C statements, a macro command ends with a newline, not a semicolon.
  - to span a macro over more than one line, enter a backslash immediately before the newline
- The simplest and mopst common use of macros is to represent numeric constant values.
  - It is also possible to create function liek macros

```
#define LONG_MACRO “This is a very long macro that \
spans two lines”
```

MACRO SUBSTITUTION

- All preprocessor directives begin with a pound sign (#), which must be the first nonspace character on the line
- Unlike C statements, a macro command ends with a newline, not a semicolon.
  - to span a macro over more than one line, enter a backslash immediately before the newline

```
#define BUFF_LEN  (512)

char buf[BUFF_LEN];

char buf[(512)];
```

FUNCTION LIKE MACROS

- **Be careful not to use**
  - ';' at the end of macro
  - or '=' in macro definition
- No type checking for macro arguments
- Try to expand min macro example for three numbers

Example 1 :

```
#define MUL_BY_TWO(a)  ((a) + (a))

j = MUL_BY_TWO(5);
f = MUL_BY_TWO(2.5);
```

Example 2 :

```
#define min(a, b) ( (a) < (b) ? (a) : (b))
```

SIDE EFFECT

```
#define min(a,b)  ((a) < (b) ? (a) : (b))

a = min(b++, c);

a =  ((b++) < (c) ? (b++) : (c));
```

- Remember min macro
- Suppose, for instance, that we invoked the ***min macro*** like this!
- The preprocessor translates this into !

MACROS VS. FUNCTIONS

Advantages

- Macros are usually faster than functions, since they avoid the **function call overhead**.
- No type restriction is placed on arguments so that one macro **may serve for several data types.**

Disadvantages

- Macro arguments are reevaluated at each mention in the macro body, which can lead to unexpected behavior if an argument contains side effects!
- Function bodies are compiled once so that multiple calls to the same function can share the same code. Macros , on the other hand, are expanded each time they appear in a program.
- Though macros check the number of arguments, they don't check the argument types.
- It is more difficult to debug programs that contain macros, because the source code goes through an additional layer of translation.

REMOVING A MACRO DEFINITION

- Once defined a macro name retains its meaning until the end of the source file.
  - or until it is expilicitly removed with an ***#undef*** directive.
- The most typical use of ***#undef*** is to remove a definition so you can **redefine** it.

```
ifndef FALSE
    define FALSE 0
elif FALSE
    undef FALSE
    define FALSE 0
endif
```

BUILT-IN MACROS

- __LINE__
  - expands to the source file line number on which it is invoked.
- __FILE__
  - expands to the name of the file in which it is invoked.
- __TIME__
  - expands to the time of program compilation.
- __DATE__
  - expands to the date of program compilation.
- __STDC__
  - Expands to the constant 1, if the compiler conforms to the ANSI Standard.

BUILT-IN MACROS

```
void print_version( ) {

    printf("This utility compiled on %s at %s\n",
                __DATE__, __TIME__);

}
```

```
void print_version( ) {

    printf("This meesage is at %d line in %s\n",
                __LINE__, __FILE__);

}
```

CONDITIONAL COMPILATION

- The preprocessor enables you to screen out portions of source code that you do not want compiled.
  - This is done through a set of preprocessor directives that are similar to ***if*** and ***else*** statements.
- The preprocessor versions are
  - #if, #else, #elif, #endif
- Conditional compilation particularly useful during the debugging stage of program development, since you can turn sections of your code on or off by changing the value of a macro
  - Most compilers have a command line option that lets you define macros before compilation begins.
  - gcc –DDEBUG=1 test.c

CONDITIONAL COMPILATION

- The conditional expression in an #if or #elif statement **need not be** enclosed in parenthesis.
- Blocks of statements under the control of a conditional preprocessor directive **are not enclosed** in braces.
- Every #if block may contain **any number** of #elif blocks, but **no more than one** #else block, which should be **the last one!**
- Every #if block must end with an #endif directive!

```
#if    x==1
       #undef   x
       define   x   0
#elif  x == 2
       #undef   x
       #define  x   3
#else
       #define  y   4
#endif
```

CONDITIONAL COMPILATION

#if defined TEST	#if defined (TEST)
#if defined *macro_name*	#ifdef *macro_name*
#if !defined macro_name	#ifndef *macro_name*

INCLUDE FACILITY

- The #include command has two forms
  - #include <filename> : the preprocessor looks in a list of implementation-defined places for the file. In UNIX systems, standard include files are often located in the directory ***/usr/include***
  - #include "filename" : the preprocessor looks for the file according to the file specification rules of operating system. If it can not find the file there, it searches for the file as if it had been enclosed in angle brackets.
- The #include command enables you to create common definition files, called header files, to be shared by several source files.
  - Traditionally have a .h extention
  - contain data structure definitions, macro definitions, function prototypes and global data

LINE CONTROL

- Allows you to change compiler's knowledge of the current line number of the source file and the name of the source file.
- The #line feature is particularly useful for programs that produce C source text.
  - for example ***yacc*** (Yet Another Compiler Compiler) is a UNIX utility that facilitates building compilers.

```
main()  {

#line 100
printf("Current line :%d\nFilename : %s\n\n",
                __LINE__, __FILE__);

#line 200  "new name"
printf("Current line :%d\nFilename : %s\n\n",
                __LINE__, __FILE__);

}
```

SPLITTING YOUR C PROGRAM

- At least one of the files must have a main() function.
- To use functions from another file,
  - make a .h file with the function prototypes,
  - and use #include to include those .h files within your .c files.
- Be sure no two files have functions with the same name in it.
  - The compiler will get confused.
- Similarly, if you use global variables in your program, be sure no two files define the same global variables.

SPLITTING YOUR C PROGRAM

- If you use global variables, be sure only one of the files defines them, and declare them in your .h as follows:
  - **extern int globalvar;**
- When you define a variable, it looks like this:
  - **int globalvar;**
- When you declare a variable, it looks like this:
  - **extern int globalvar;**
- The main difference is that a variable definition creates the variable, while a declaration indicates that the variable is defined elsewhere. A definition implies a declaration.

AN EXAMPLE

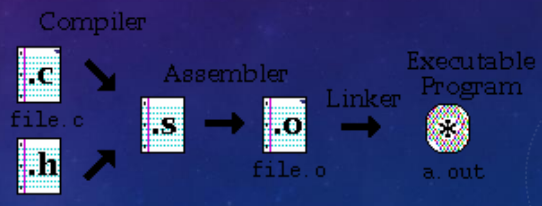
```
#ifndef __salute_h__
#define __salute_h__
void salute( void );
#endif
```

```
#include <stdio.h>

void salute( void ) {
        printf("\n\n HELLO!!! \n\n");
}
```

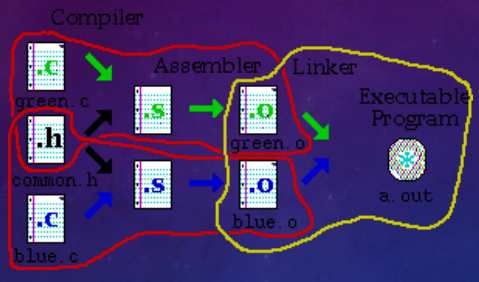
```
#include <stdio.h>
#include "salute.h"
int main ( void ) {

        salute();
        return 0;
}
```

COMPILING WITH SEVERAL FILES

- The command to perform this task is simply
  - **gcc file.c**
- There are 3 steps to obtain the final executable program
  - **Compiler stage**
  - **Assembler stage**
  - **Linker stage**



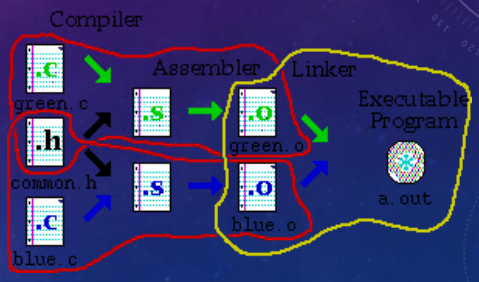
COMPILING WITH SEVERAL FILES



- You can use the -c option with gcc to create the corresponding object (.o) file from a .c file.
  - **gcc -c green.c**
- will not produce an a.out file, but the compiler will stop after the assembler stage, leaving you with a green.o file.

COMPILING WITH SEVERAL FILES

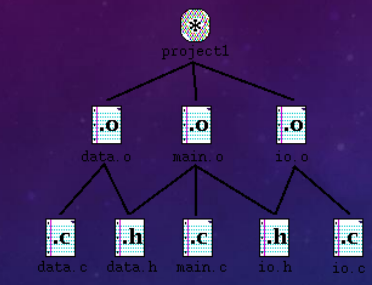
- The three different tasks required to produce the executable program are as follows:
- Compile green.o:
  - **gcc -c green.c**
- Compile blue.o:
  - **gcc -c blue.c**
- Link the parts together:
  - **gcc green.o blue.o**



THE MAKE COMMAND

- helps you to manage large programs or groups of programs
- keeps track of which portions of the entire program have been changed
- compiles only those parts of the program which have changed since the last compile.

HOW DOES MAKE DO IT?

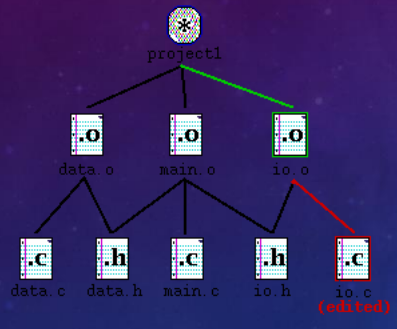


Sample Makefile

```
project1: data.o main.o io.o
     cc data.o main.o io.o -o project1
data.o: data.c data.h
     cc -c data.c
main.o: data.h io.h main.c
     cc -c main.c
io.o: io.h io.c
     cc -c io.c
```

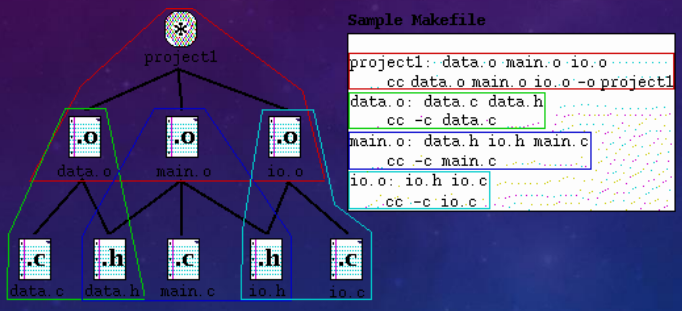
- The make program gets its dependency "graph" from a text file called makefile or Makefile, which resides in the same directory as the source files
- make checks the modification times of the files, and whenever a file becomes "newer" than something that depends on it, it runs the compiler accordingly.

HOW DEPENDENCY WORKS



- Case : while you are testing the program, you realize that one function in io.c has a bug in it. You edit io.c to fix the bug.
- notice that io.o needs to be updated because io.c has changed. Similarly, because io.o has changed, project1 needs to be updated as well.

TRANSLATING THE DEPENDENCY GRAPH



- Each dependency shown in the graph is circled with a corresponding color in the Makefile, and each uses the following format:
- target : source file(s)
  - command (must be preceded by a tab)

LISTING DEPENDENCIES

- Note that in the Makefile shown on the right, the .h files are listed, but there are no references in their corresponding commands.
- This is because the .h files are referred within the corresponding .c files through the #include "file.h".
- If you do not explicitly include these in your Makefile, your program will not be updated if you make a change to your header (.h) files.

Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

USING THE MAKEFILE WITH MAKE

- Once you have created your Makefile and your corresponding source files, you are ready to use make.
- If you have named your Makefile either Makefile or makefile, make will recognize it.
- If you do not wish to call your Makefile one of these names, you can use make -f mymakefile.

MACROS IN MAKE

- The make program allows you to use macros, which are similar to variables, to store names of files. The format is as follows:
  - OBJECTS = data.o io.o main.o
- Whenever you want to have make expand these macros out when it runs, type the following corresponding string $(OBJECTS)

Here is our sample *Makefile* again, using a macro.

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

SPECIAL MACROS

- In addition to those macros which you can create yourself, there are a few macros which are used internally by the make program. Here are some of those, listed below:

CC	Contains the current C compiler. Defaults to cc.
CFLAGS	Special options which are added to the built-in C rule.
$@	Full name of the current target.
$?	A list of files for current dependency which are out-of-date.
$<	The source file of the current (single) dependency.

REFERENCES & MORE READING

- References
  - http://www.eng.hawaii.edu/Tutor/Make/index.html
- More reading on make command
  - http://www.cs.duke.edu/~ola/courses/programming/Makefiles/Makefiles.html
  - http://www.hsrl.rutgers.edu/ug/make_help.html
  - http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/make.htm