C PREPROCESSOR

Outline

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

The C Preprocessor

- The C preprocessor as a separate program that runs before the compiler
- It has its own simple, line-oriented grammar and syntax.
- Example for two preprocessor directives:
  - **#define** command for naming a constant and
  - **#include** command for including additional source files.
- We will describe other preprocessor directives
- Briefly, the preprocessor gives you the following capabilities:
  - Macro processing.
  - Inclusion of additional C source files.
  - Conditional compilation

Macros

- All preprocessor directives begin with a pound sign (#), which must be the first nonspace character on the line (some compilers may not allow spaces)
- They may appear anywhere in the source file - before, after, or intermingled with regular C language statements.
- Unlike C statements, a macro command ends with a newline, not a ";".
  - To span a macro over more than one line, enter a backslash (\) immediately before the newline
- The simplest and most common use of macros is to represent numeric constant values.
  - It is also possible to create functions like macros

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

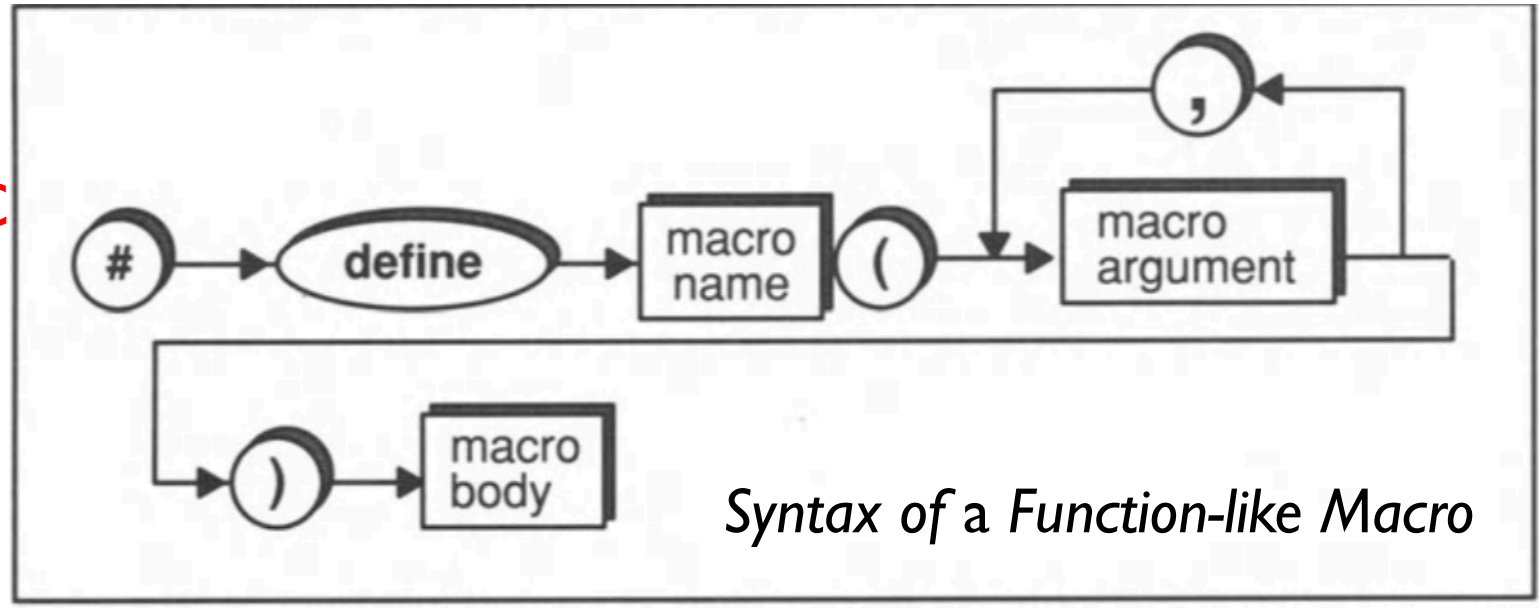
Macro substitution

- A *macro* is a name that has an associated text string, called the *macro body*.
- By convention, macro names that represent constants should consist of uppercase letters only.
- This makes it easy to distinguish macro names from variable names
- When a macro name appears outside its definition it is replaced with its macro body. The act of replacement is referred to as *macro expansion*.
- In the following example, *BUF_LEN* is the macro name and 512 is the macro body.

```
#define BUFF_LEN   512
char buf[BUFF_LEN];
char buf[512];
```

- The simplest and most common use of macros is to represent numeric constant values.
- As with choosing names for variables, it is important to choose a macro name that corresponds to its use.
- #define MAX_INPUT_BUFFER_SIZE 256

There is another form of macros that is similar to a C function in that it takes *arguments* that can be used in the macro body



Syntax of a Function-like Macro

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))
// it reduces a * operation into an + operation, which is faster. And "a" can be int or float
j = MUL_BY_TWO(5);
f = MUL_BY_TWO(2.5);
```

Example 2 :

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

- The parentheses around *a* and *b* and around the *macro body* are necessary to ensure correct binding when the macro is expanded

Function-like macros

- Macro arguments are not variables, they have no type, and no storage is allocated for them.
- So, macro arguments do not conflict with variables that have the same name.
- The following, for example, is perfectly legal:

```
j = MUL_BY_TWO(a-1);
```

which, after expansion, becomes

```
j = ((a-1) + (a-1));
```

- In general, macros execute more quickly than functions because there is none of the function overhead involved in *copying arguments* and *maintaining stack frames*.
- When trying to speed up slow programs, therefore, you should be on the lookout for small, heavily used functions that can be implemented as macros.
- A function that converts a letter from uppercase to lowercase. Assuming an ASCII character set, we can rewrite it as:
- #define TO_LOWER(e) ((e) + ('a' - 'A'))

SIDE EFFECT

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

z = min(x++, y);

z =  ((x++) < (y) ? (x++) : (y));
```

If b<c, it gets incremented twice, obviously not what is intended.

To be on the safe side, you should never use a side effect operator (++, --, =, *function invocation*) in a macro.

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

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 *(you cannot easily write generic functions that can work for all data types, as macros can do).***

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.

Box 10-3: Bug Alert — Using = to Define a Macro

A common mistake made in defining macros is to use the assignment operator as if you were initializing a variable. Instead of writing

```
#define MAX 100
```

you write

```
#define MAX = 100
```

This type of mistake can lead to obscure bugs. For example, the expression

```
for (j=MAX; j > 0; j--)
```

would expand to

```
for (j== 100; j > 0; j--)
```

Suddenly, the assignment is turned into a relational expression. The expression is legal, so the compiler will not complain, making the error difficult to track down.

Box 10-7: Bug Alert — Binding of Macro Arguments

A potential problem with macros is that argument expressions that are not carefully parenthesized can produce erroneous results due to operator precedence and binding. Consider the following macro:

```
#define square( a ) a * a
```

square has the advantage that it will work regardless of the argument data types. However, watch what happens when we pass it an arithmetic expression:

```
j = 2 * square( 3 + 4 );
```

expands to

```
j = 2 * 3 + 4 * 3 + 4;
```

Because of operator precedence, the compiler interprets this expression as

```
j = (2 * 3) + (4 * 3) + 4;
```

which assigns the value of 22 to *j*, instead of 98. To avoid this problem, you should always enclose the macro body and macro arguments in parentheses:

```
#define square( a ) ((a) * (a))
```

Now, the macro invocation expands to

```
j = 2 * ((3 + 4) * (3 + 4));
```

which produces the correct result.

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.

- #undef min

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.

- The *_LINE_ and _FILE_ macros* are available in most older compilers.
- The *_TIME_*, *_DATE_*, and *STDC* are more recent ANSI additions to the C preprocessor.
- The *_LINE_* and *_FILE_* macros are valuable diagnostic tools.

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__);
 }
```

built-in macros - example

```
#define CHECK( a, b ) \
if((a)!=(b)) \
fail( a, b, __FILE__, __LINE__)

void fail(int a, int b, char* p, int line)
{
   printf( "Check failed in file %s at line %d:\ received %d, expected
        %d\n", p, line, a, b );
}
```

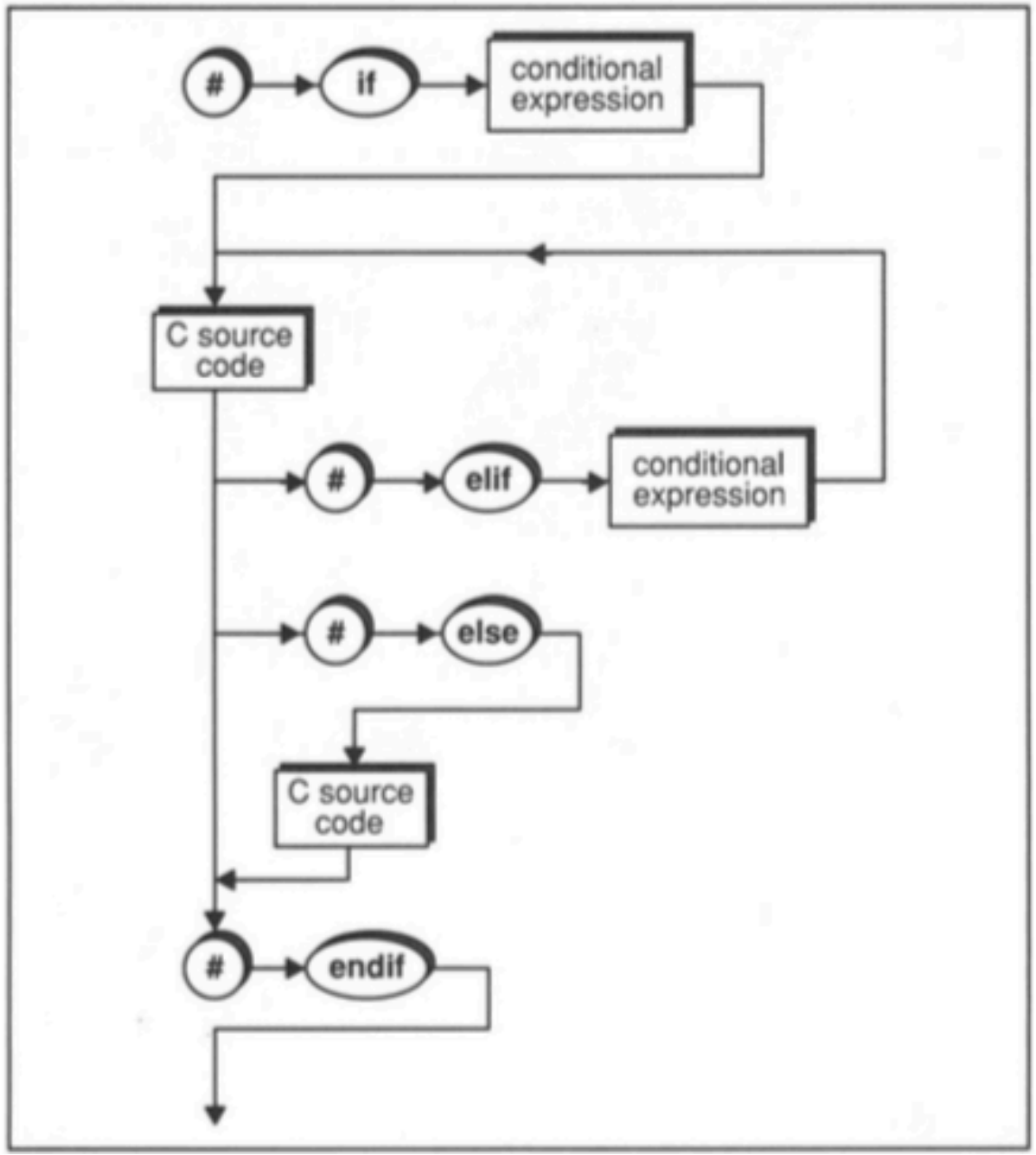
- At various points in a program, you can check to make sure that a variable *x* equals zero by including the following diagnostic:

```
CHECK(x, 0); // check the code shown in the class
```

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

Syntax of Conditional Compilation Directives



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
```

- In the earlier example, the statements within the conditional blocks are themselves preprocessor statements, but this is not a restriction.
  - They could just as easily be C language statements.
- Conditional compilation is particularly useful during the debugging stage of program development since you can turn sections of code on or off by changing the value of a macro.
- #if DEBUG

```
if(exp_debug){
   printf("…\n");
   func1(…);
}
#endif
```

- Most compilers have a command line option that lets you define macros before compilation begins: "**-D** option for defining macros".
- To receive debug information, you would define the macro *DEBUG* to be some non-0 value:

```
cc -DDEBUG=1  test.c
```

Conditional compilation - testing macro existence

Equivalent

#if defined TEST

#if defined *macro_name*

#if !defined macro_name

#if defined (TEST)

#ifdef *macro_name*

#ifndef *macro_name*

```
#ifdef  TEST
    printf("This is a test. \n");
#else
    printf("This is not a test. \n");
#endif
```

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

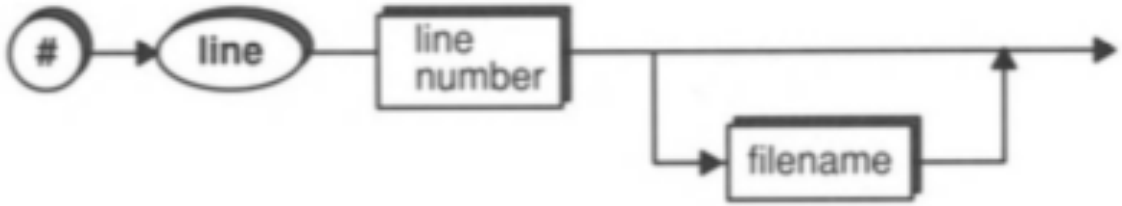
include facility

- 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 extension
  - 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.

```
#include <stdio.h>
main()  {
printf("Current line :%d\nFilename : %s\n\n",    //Current line: 4
    __LINE__, __FILE__);  //Filename: line_example.c
#line 100
printf("Current line :%d\nFilename : %s\n\n",    //Current line:101
    __LINE__, __FILE__); // Filename: line_example.c
#line 200  "new name"
printf("Current line :%d\nFilename : %s\n\n",  // Current line: 201
    __LINE__, __FILE__);  //Filename: new name
}
```

Box 10-10: ANSI Feature — The *#error* Directive

The ANSI **#error** directive enables you to report errors during the preprocessing stage of compilation. Whatever text follow the **#error** command will be sent to the standard error device (usually your terminal). Typically, it is used to check for illegal conditional compilation values. For example,

```
#if INTSIZE < 16
#  error INTSIZE too small
#endif
```

If you attempt to compile a file with

```
cc -DINTSIZE=8 test.c
```

you will receive the error message

```
INTSIZE too small
```

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 2 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

salute.h (header file contains prototypes):

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

salute.c (implement the .h in a .c file):

```
#include <stdio.h>
void salute( void )  {
     printf("\n\n HELLO!!! \n\n");
}
```

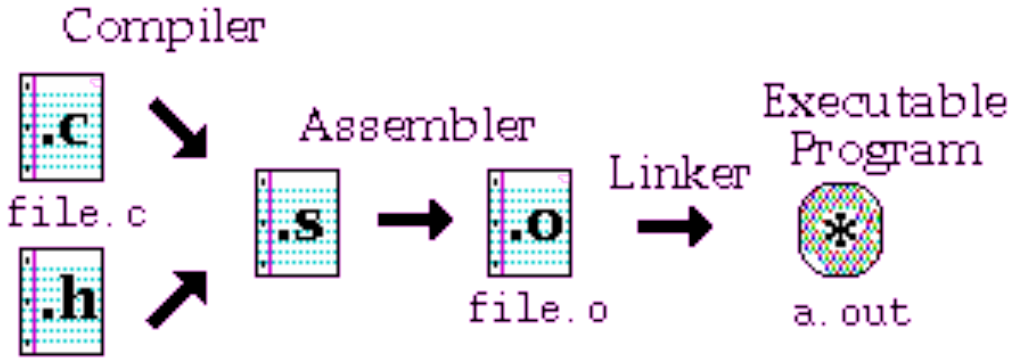
main.c :

```
#include <stdio.h>
#include "salute.h"
int main ( void )  {
          salute();
          return 0;
}
```

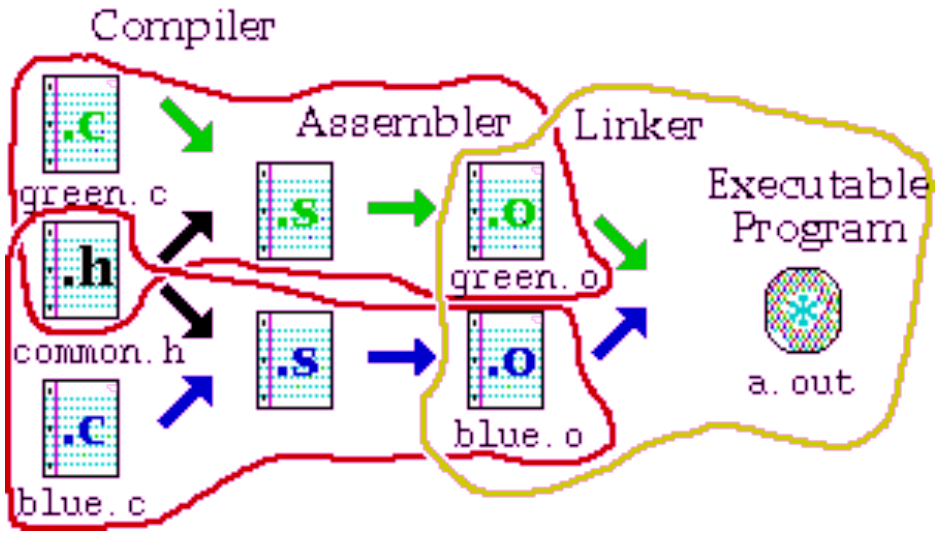
```
cc -c salute.c #this will give you salute.o
cc -c main.c #this will give you main.o
cc -o main main.o salute.o #this will create the main executable
```

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**
- Side-note: pre-processor takes place before the compiler stage



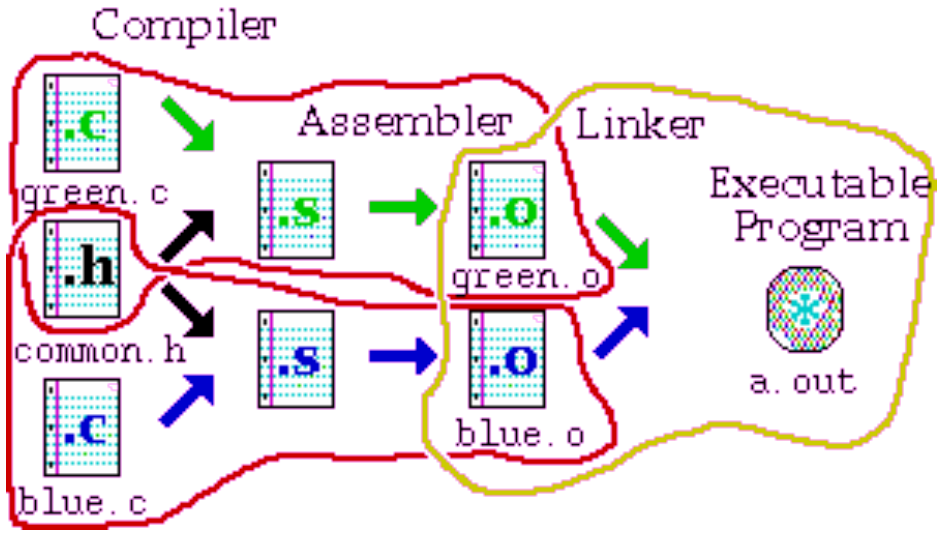
Compiling with Several Files



- 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

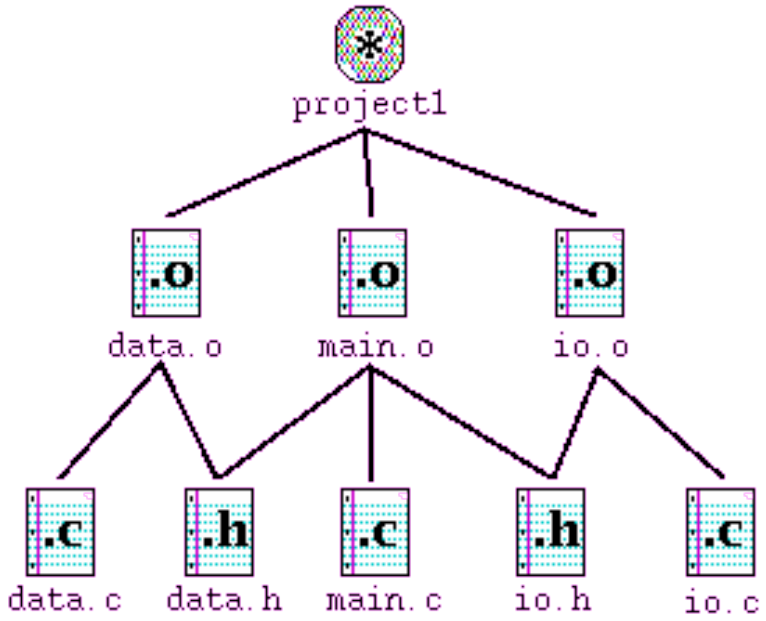
- The three different tasks required to produce the executable program are as follows:
- Compile green.o:
  - **gcc -c green.c**
- Compile blue.o:
  - **gcc -c blue.c**
- Link the parts together:
  - **gcc green.o blue.o** // will create an exe with the name ***a.out*** (by default)
  - **If you want to give another name to your exe, use –o flag**



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?



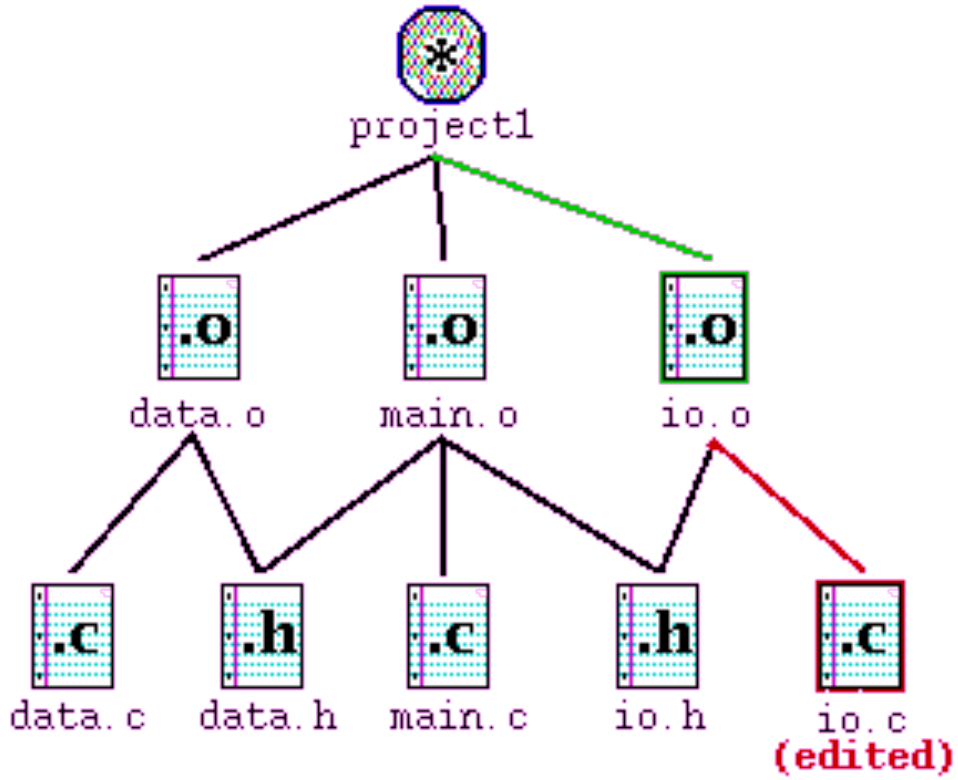
Sample Makefile

```
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

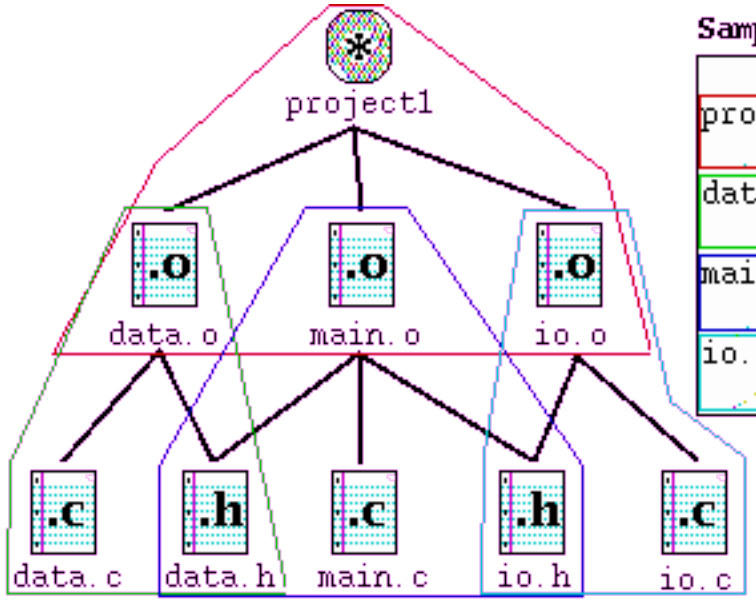
make checks the modification times of the files, and whenever a file becomes "newer" than something that depends on it, it runs the compiler accordingly.

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



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
```

- 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* command 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:
  - OBJECTS = data.o io.o main.o
- Whenever you want to have make expand these macros out when it runs, type the 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, which are used by the make program

- 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