Introduction to Computers and Programming

Lecture 8 –
Preprocessing and multiple files for large program

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Preprocessing directives

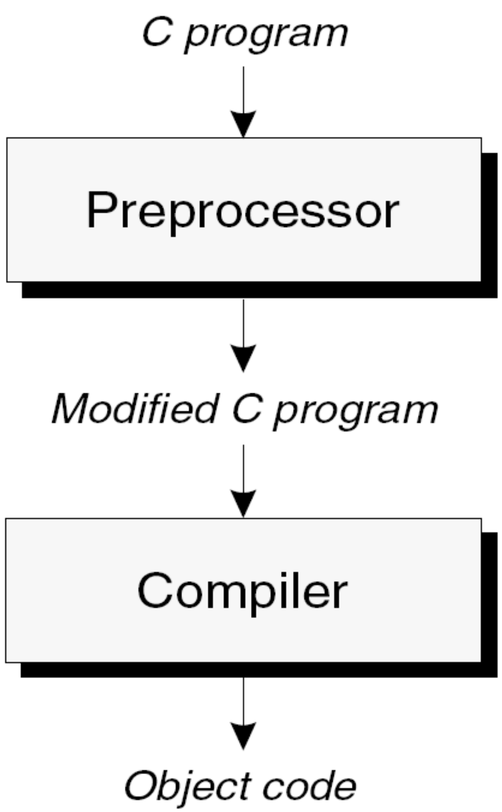
- ***Preprocessing directives*** begin with a `#` character.
- `#define` defines a ***macro***—
  - Preprocessor handle `#define` directive by storing the name of the macro along with its definition.
  - When the macro is used later, the preprocessor "expands" the macro, replacing it by its defined value.
- `#include` to open a particular file and "include" its contents as part of the file being compiled.

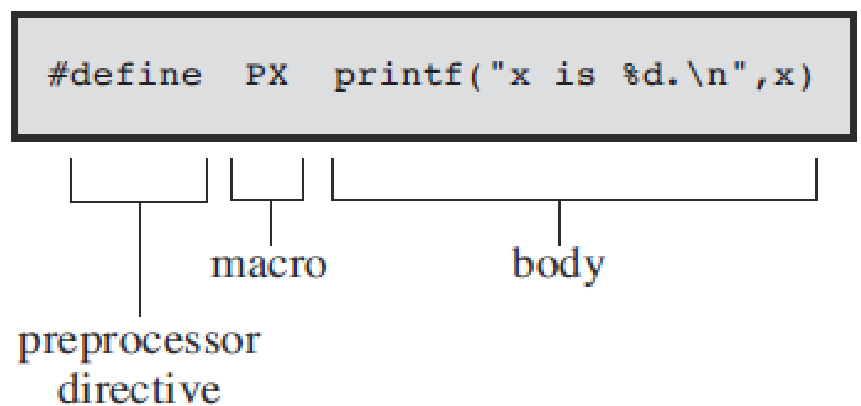
```
#include <stdio.h>
```

  instructs the preprocessor to open the file named `stdio.h` and bring its contents into the program.

How the Preprocessor Works

- The preprocessor's role in the compilation process:





How the Preprocessor Works

- The `celsius.c` program :

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
  printf("Celsius equivalent is: %.1f\n", celsius);

}
```

How the Preprocessor Works

- The program after preprocessing:

Blank line
Blank line
Lines brought in from stdio.h
Blank line
Blank line
Blank line
Blank line

```
int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);

  printf("Celsius equivalent is: %.1f\n", celsius);
}
```

Three directive categories

- ***Macro definition.***

The `#define` directive defines a macro; the `#undef` directive removes a macro definition.

- ***File inclusion.***

The `#include` directive causes the contents of a specified file to be included in a program.

- ***Conditional compilation.***

The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program.

Preprocessing Directives

- ***Directives always end at the first new-line character, unless explicitly continued.***

  To continue a directive to the next line, end the current line with a \ character:

```
#define DISK_CAPACITY (SIDES *              \
                       TRACKS_PER_SIDE *    \
                       SECTORS_PER_TRACK *  \
                       BYTES_PER_SECTOR)
```

Simple Macros

- Any extra symbols in a macro definition will become part of the replacement list.
- Putting the = symbol in a macro definition is a common error:

```
#define N = 100   /*** WRONG ***/
…
int a[N];         /* becomes int a[= 100]; */


#define N 100;   /*** WRONG ***/
…
int a[N];        /* becomes int a[100;]; */
```

Simple Macros

- Simple macros are primarily used for defining "manifest constants"—names that represent numeric, character, and string values:

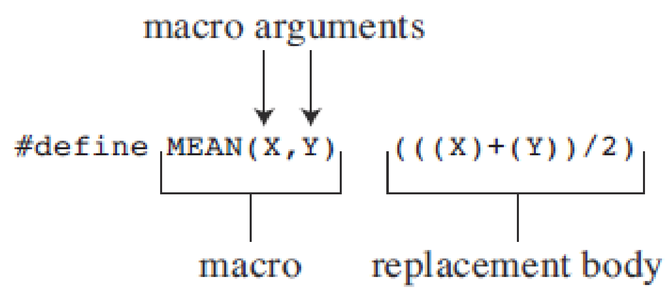
```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
#define MEM_ERR "Error: not enough memory"
```

Parameterized Macros

- Definition of a ***parameterized macro*** (also known as a ***function-like macro***):

  `#define` *identifier*`(` x_1 `,` x_2 `,` … `,` x_n `)` *replacement-list*

  x_1, x_2, …, x_n are the macro's ***parameters***.



- There must be *no space* between the macro name and the left parenthesis.
  - If space is left, the preprocessor will treat (x_1, x_2, …, x_n) as part of the replacement list.

Parameterized Macros

- Examples of parameterized macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)
```

- Invocations of these macros:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

- The same lines after macro replacement:

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

Parameterized Macros

- A more complicated function-like macro:

```
#define TOUPPER(c) \
   ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

- A parameterized macro may have an empty parameter list:

```
#define getchar() getc(stdin)
```

Parameterized Macros

- Parameterized macros also have disadvantages.

  The compiled code will often be larger.

  Each macro invocation increases the size of the source program (and hence the compiled code).

when macro invocations are nested:

```
n = MAX(i, MAX(j, k));
```

The statement after preprocessing:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

Parameterized Macros

- ***A macro may evaluate its arguments more than once.***

  Unexpected behavior may occur if an argument has side effects:

  ```
  n = MAX(i++, j);
  ```

  The same line after preprocessing:

  ```
  n = ((i++)>(j)?(i++):(j));
  ```

  If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

Parameterized Macros

- Parameterized macros can be used as patterns for segments of code that are often repeated.
- A macro that makes it easier to display integers:

```
#define PRINT_INT(n) printf("%d\n", n)
```

- The preprocessor will turn the line

```
PRINT_INT(i/j);
```

into

```
printf("%d\n", i/j);
```

The # Operator

- Macro definitions may contain two special operators, # and ##.
- Neither operator is recognized by the compiler; instead, they're executed during preprocessing.

- The # operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro.
- The operation performed by # is known as "stringization."

The # Operator

- Our new version of `PRINT_INT`:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

- The invocation

```
PRINT_INT(i/j);
```

  will become

```
printf("i/j" " = %d\n", i/j);
```

- The compiler automatically joins adjacent string literals, so this statement is equivalent to

```
printf("i/j = %d\n", i/j);
```

The ## Operator

- The `##` operator can "paste" two tokens together to form a single token.

- A macro that uses the `##` operator:

```
#define MK_ID(n) i##n
```

- A declaration that invokes `MK_ID` three times:

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

- The declaration after preprocessing:

```
int i1, i2, i3;
```

General Properties of Macros

- ***A macro's replacement list may contain invocations of other macros.***

  Example:

  ```
  #define PI      3.14159
  #define TWO_PI (2*PI)
  ```

  When it encounters `TWO_PI` later in the program, the preprocessor replaces it by `(2*PI)`.

General Properties of Macros

- ***Macros may be "undefined" by the `#undef` directive.***

  The `#undef` directive has the form

  `#undef` *identifier*

  where *identifier* is a macro name.

  One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

Parentheses in Macro Definitions

- If the macro's replacement list contains an operator, always enclose the replacement list in parentheses:

```
#define TWO_PI (2*3.14159)
```

- Also, put parentheses around each parameter every time it appears in the replacement list:

```
#define SCALE(x) ((x)*10)
```

Parentheses in Macro Definitions

- An example that illustrates the need to put parentheses around a macro's replacement list:

```
#define TWO_PI 2*3.14159
/* needs parentheses around replacement list */
```

- During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication.

Parentheses in Macro Definitions

- Each occurrence of a parameter in a macro's replacement list needs parentheses as well:

```
#define SCALE(x) (x*10)
  /* needs parentheses around x */
```

- During preprocessing, the statement

```
j = SCALE(i+1);
```

  becomes

```
j = (i+1*10);
```

  This statement is equivalent to

```
j = i+10;
```

Creating Longer Macros

- An alternative definition of `ECHO` that uses braces:

```
#define ECHO(s) { gets(s); puts(s); }
```

- Suppose that we use `ECHO` in an `if` statement:

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
```

- Replacing `ECHO` gives the following result:

```
if (echo_flag)
  { gets(str); puts(str); };
else
  gets(str);
```

Creating Longer Macros

- A modified version of the `ECHO` macro:

```
#define ECHO(s)         \
        do {            \
          gets(s);      \
          puts(s);      \
        } while (0)
```

- When `ECHO` is used, it must be followed by a semicolon, which completes the `do` statement:

```
ECHO(str);
/* becomes
   do { gets(str); puts(str); } while (0); */
```

Predefined Macros

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format
__TIME__	The current time as a character literal in "HH:MM:SS" format
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

- Example of using `__DATE__` and `__TIME__`:

```
printf("Wacky Windows (c) 2010 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

- Output produced by these statements:

```
Wacky Windows (c) 2010 Wacky Software, Inc.
Compiled on Dec 23 2010 at 22:18:48
```

Predefined Macros

- We can use the `__LINE__` and `__FILE__` macros to help locate errors.
- A macro that can help pinpoint the location of a division by zero:

```
#define CHECK_ZERO(divisor) \
  if (divisor == 0) \
    printf("*** Attempt to divide by zero on line %d " \
           "of file %s ***\n", __LINE__, __FILE__)
```

- The `CHECK_ZERO` macro would be invoked prior to a division:

```
CHECK_ZERO(j);
k = i / j;
```

Useful assert

- **Debug.h:**

```
#define assert(EX) \
((EX)?((void)0):myassert( # EX, __FILE__, __LINE__))
```

- **Debug.c:**

```
void myassert(const char *msg, char *file, int line)
{
   fprintf (stderr,"assertion failed:"
    "%s:%d: \"%s\"\n", file, line,  msg);
}

assert(i!=0);
X = 100/i;

(i!=0)?((void)0): myassert(“i!=0”, “test.c”, 100));
```

The `#if` and `#endif` Directives

- The first step is to define a macro and give it a nonzero value:

```
#define DEBUG 1
```

- Next, we'll surround each group of `printf` calls by an `#if`-`#endif` pair:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

The `#if` and `#endif` Directives

- General form of the `#if` and `#endif` directives:

  ```
  #if constant-expression
  #endif
  ```

- When the preprocessor encounters the `#if` directive, it evaluates the constant expression.
- If the value of the expression is zero, the lines between `#if` and `#endif` will be removed from the program during preprocessing.
- Otherwise, the lines between `#if` and `#endif` will remain.

The `#if` and `#endif` Directives

- The `#if` directive treats undefined identifiers as macros that have the value 0.
- If we neglect to define `DEBUG`, the test

  ```
  #if DEBUG
  ```

  will fail (but not generate an error message).
- The test

  ```
  #if !DEBUG
  ```

  will succeed.

The `defined` Operator

- Example:

```
#if defined(DEBUG)
…
#endif
```

- The lines between `#if` and `#endif` will be included only if `DEBUG` is defined as a macro.
- The parentheses around `DEBUG` aren't required:

```
#if defined DEBUG
```

- It's not necessary to give `DEBUG` a value:

```
#define DEBUG
```

The `#ifdef` and `#ifndef` Directives

- The `#ifdef` directive tests whether an identifier is currently defined as a macro:

  `#ifdef` *identifier*

- The effect is the same as

  `#if defined(`*identifier*`)`

- The `#ifndef` directive tests whether an identifier is *not* currently defined as a macro:

  `#ifndef` *identifier*

- The effect is the same as

  `#if !defined(`*identifier*`)`

The `#elif` and `#else` Directives

- `#if`, `#ifdef`, and `#ifndef` blocks can be nested just like ordinary `if` statements.
- When nesting occurs, it's a good idea to use an increasing amount of indentation as the level of nesting grows.
- Some programmers put a comment on each closing `#endif` to indicate what condition the matching `#if` tests:

```
#if DEBUG
…
#endif /* DEBUG */
```

Uses of Conditional Compilation

- Conditional compilation has other uses besides debugging.
- ***Writing programs that are portable to several machines or operating systems.***

  Example:

```
#if defined(WIN32)
…
#elif defined(MAC_OS)
…
#elif defined(LINUX)
…
#endif
```

Uses of Conditional Compilation

- ***Providing a default definition for a macro.***

  Conditional compilation makes it possible to check whether a macro is currently defined and, if not, give it a default definition:

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

Uses of Conditional Compilation

- ***Temporarily disabling code that contains comments.***

  A `/*…*/` comment can't be used to "comment out" code that already contains `/*…*/` comments.

  An `#if` directive can be used instead:

```
#if 0
```

  Lines containing comments

```
#endif
```

Large program by multiple files

Source Files

- Splitting a program into multiple source files has significant advantages:
  - Grouping related functions and variables into a single file helps clarify the structure of the program.
  - Each source file can be compiled separately, which saves time.
  - Functions are more easily reused in other programs when grouped in separate source files.

The `#include` Directive

- The `#include` directive has two primary forms.
- The first is used for header files that belong to C's own library:

  #include <*filename*>

- The second is used for all other header files:

  #include "*filename*"

- The difference between the two has to do with how the compiler locates the header file.

The `#include` Directive

- It's usually best not to include path or drive information in `#include` directives.
- Bad examples of Windows `#include` directives:

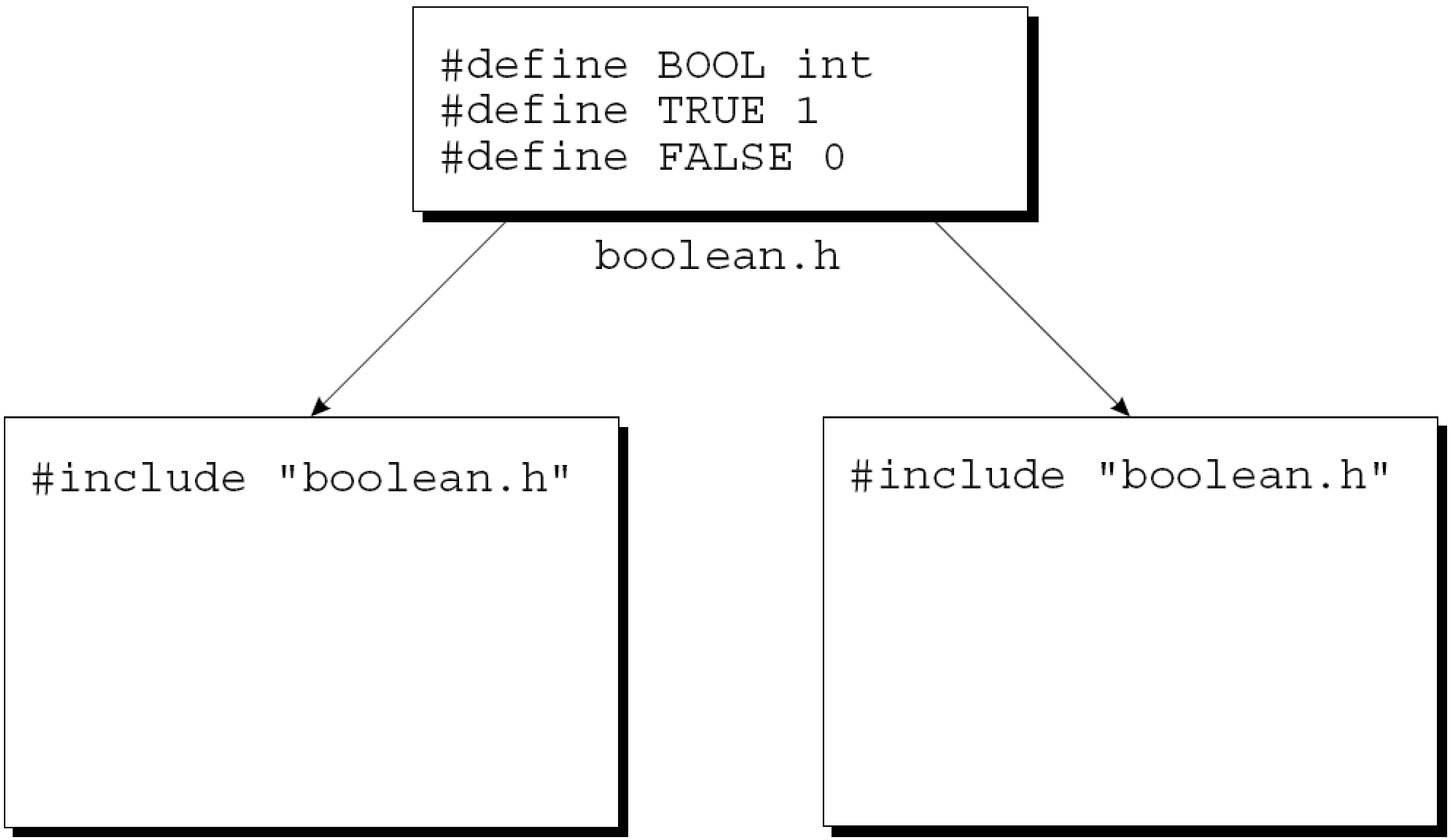
```
#include "d:utils.h"
#include "\cprogs\include\utils.h"
#include "d:\cprogs\include\utils.h"
```

- Better versions:

```
#include "utils.h"
#include "..\include\utils.h"
```

Sharing Macro Definitions and Type Definitions

- A program in which two files include `boolean.h`:



Sharing Function Prototypes

- The `stack.c` file will contain definitions of the `make_empty`, `is_empty`, `is_full`, `push`, and `pop` functions.
- Prototypes for these functions should go in the `stack.h` header file:

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

Sharing Function Prototypes

```c
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

stack.h

```c
#include "stack.h"

int main(void)
{
  make_empty();
  …
}
```

calc.c

```c
#include "stack.h"

int contents[100];
int top = 0;

void make_empty(void)
{ … }

int is_empty(void)
{ … }

int is_full(void)
{ … }

void push(int i)
{ … }

int pop(void)
{ … }
```

stack.c

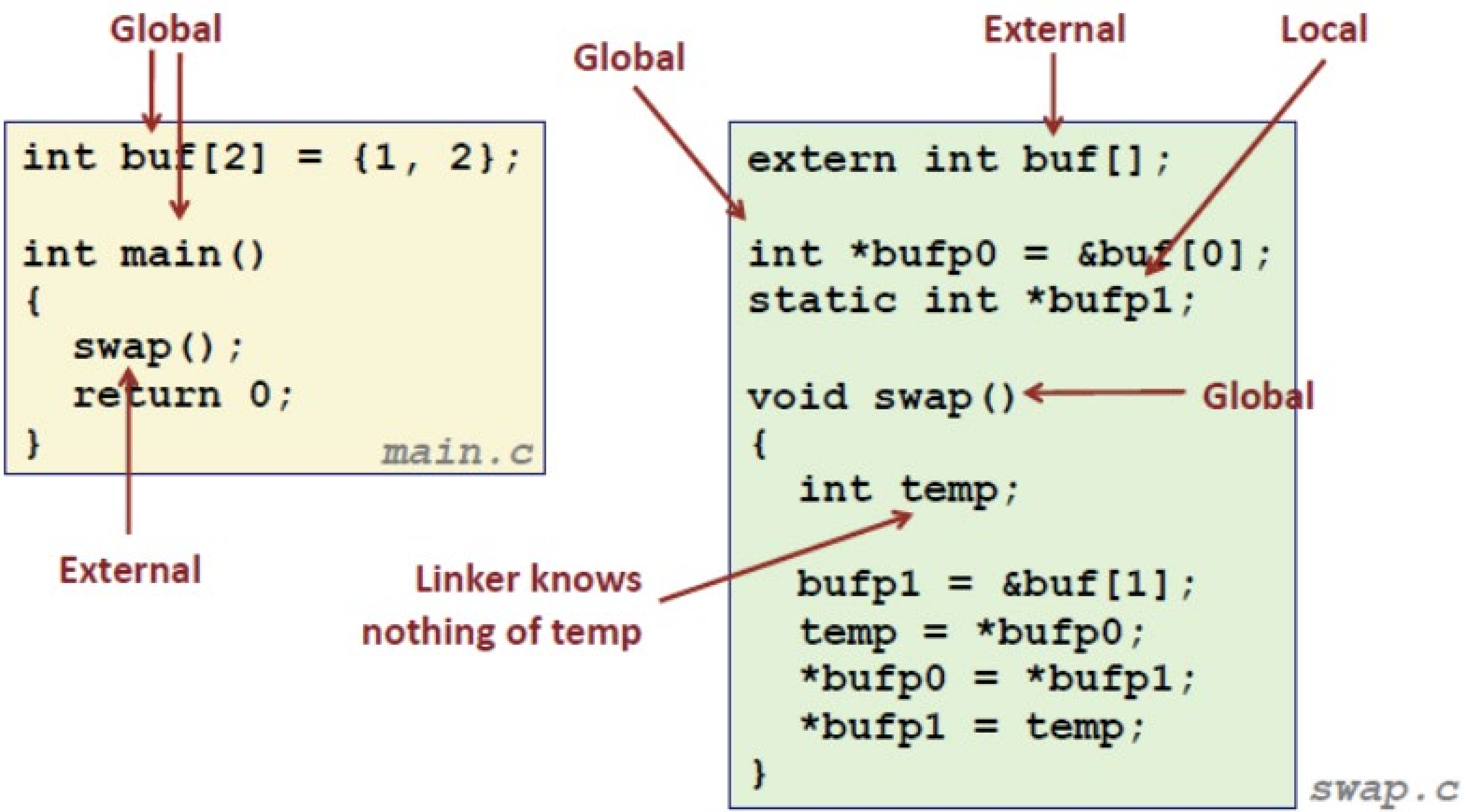
Sharing Variable Declarations

- We first put a definition of `i` in one file:

```
int i;
```

- The other files will contain declarations of `i`:

```
extern int i;
```



Nested Includes

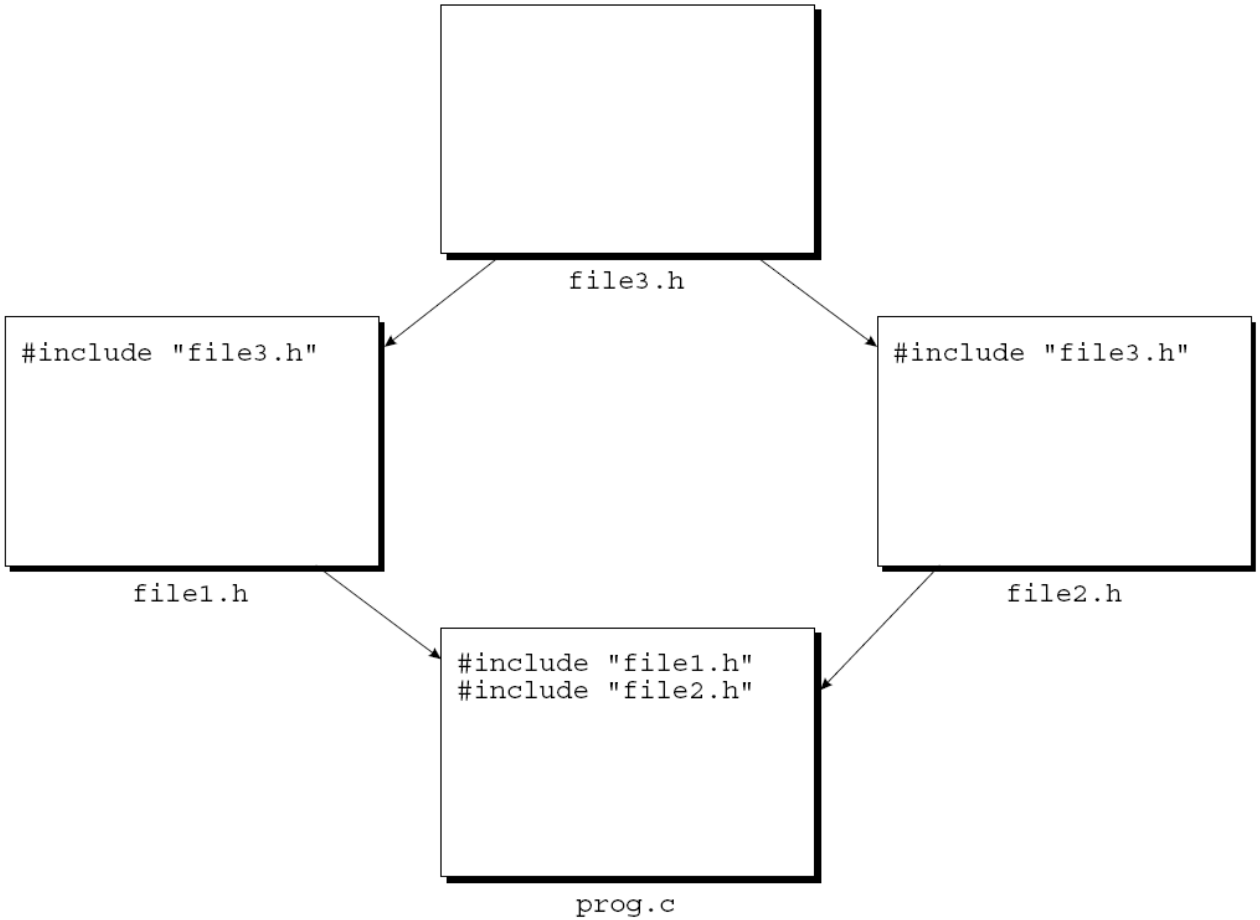
- A header file may contain `#include` directives.
- `stack.h` contains the following prototypes:

```
int is_empty(void);
int is_full(void);
```

- Since these functions return only 0 or 1, it's a good idea to declare their return type to be `Bool`:

```
Bool is_empty(void);
Bool is_full(void);
```

Protecting Header Files



- When `prog.c` is compiled, `file3.h` will be compiled twice.

Protecting Header Files

- To protect a header file, we'll enclose the contents of the file in an `#ifndef`-`#endif` pair.
- How to protect the `boolean.h` file:

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```

The #include Directive

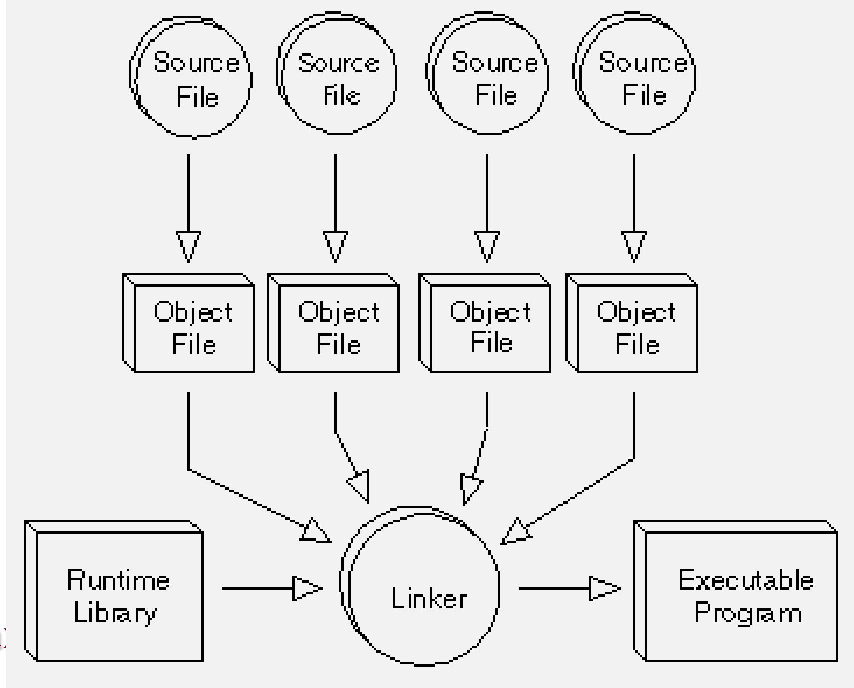
- Example:

```
#if defined(IA32)
  #define CPU_FILE "ia32.h"
#elif defined(IA64)
  #define CPU_FILE "ia64.h"
#elif defined(AMD64)
  #define CPU_FILE "amd64.h"
#endif

#include CPU_FILE
```

Building a Multiple-File Program

- Each source file must be compiled separately.
- Header files don't need to be compiled.
- The contents of a header file are automatically compiled whenever it is included.
- For each source, the compiler generates its object code, having the extension `.o` in UNIX and `.obj` in Windows.

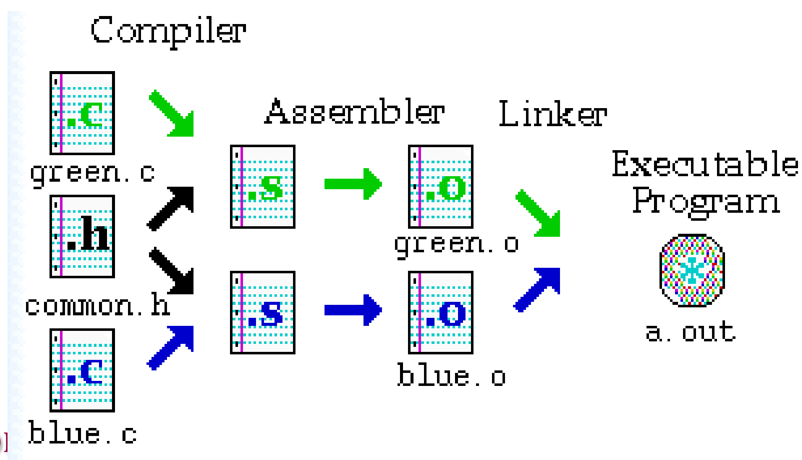


Building a Multiple-File Program

- A GCC command that builds `justify`:

```
gcc -o justify justify.c line.c word.c
```

- The three source files are first compiled into object code.
- The object files are then automatically passed to the linker, which combines them into a single file.
- The `-o` option specifies that we want the executable file to be named `justify`.



Makefile

- A makefile not only lists the files that are part of the program, but also describes ***dependencies*** among the files.
- the file `foo.c` includes the file `bar.h`.
  - `foo.c` "depends" on `bar.h`, because a change to `bar.h` will require us to recompile `foo.c`.

target

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
        gcc -c justify.c

word.o: word.c word.h
        gcc -c word.c

line.o: line.c line.h
        gcc -c line.c
```

rule

command

Rebuilding a Program

- If the change affects a single source file, only that file must be recompiled.
- Suppose to condense the `read_char` function in `word.c`:

```
int read_char(void)
{
  int ch = getchar();
  return (ch == '\n' || ch == '\t') ? ' ' : ch;
}
```

- This modification doesn't affect `word.h`, so we need only recompile `word.c` and relink the program.

Defining Macros Outside a Program

- Most compilers (including GCC) support the `-D` option, which allows the value of a macro to be specified on the command line:

```
gcc -DDEBUG=1 foo.c
```

- the `DEBUG` macro is defined to have the value `1` in the program `foo.c`.
- If the `-D` option names a macro without specifying its value, the value is taken to be `1`.