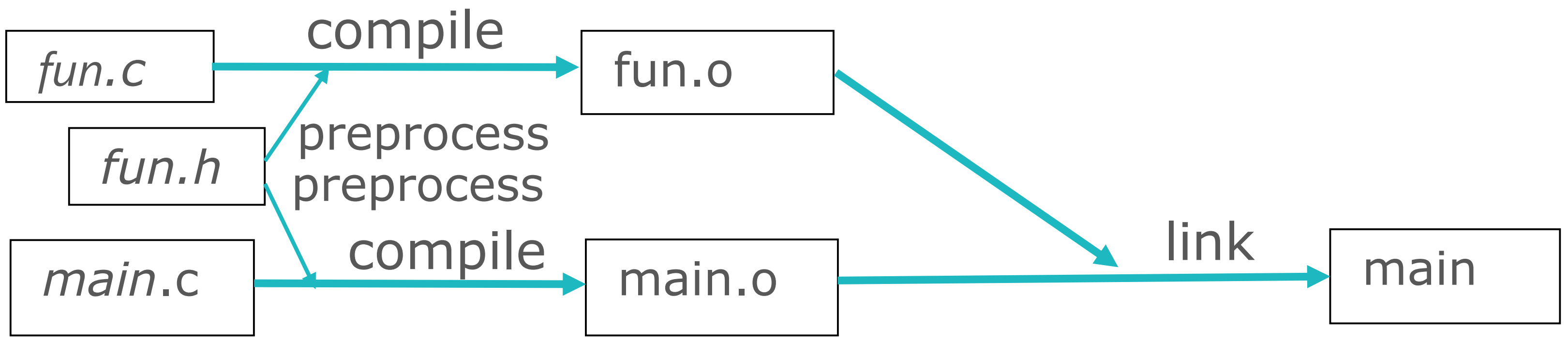
Lecture 15 - Writing Large Programs

Meng-Hsun Tsai
CSIE, NCKU



15.1 Source Files and Header Files

Source Files

- A C program may be divided among any number of ***source files***.
- By convention, source files have the extension `.c`.
- Each source file contains part of the program, primarily definitions of functions and variables.
- One source file must contain a function named `main`, which serves as the starting point for the program.

Source Files (cont.)

- 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.

Header Files

- Problems that arise when a program is divided into several source files:
  - How can a function in one file call a function that's defined in another file?
  - How can a function access an external variable in another file?
  - How can two files share the same macro definition or type definition?
- The answer lies with the `#include` directive, which makes it possible to share information among any number of source files.

Header Files (cont.)

- Files that are included through `#include` are called ***header files*** (or sometimes ***include files***).
- By convention, header files have the extension `.h`.
- 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 `#include` Directive

- Typical rules for locating header files:
  - `#include` <*filename*>: Search the directory (or directories) in which system header files reside.
  - `#include` "*filename*": Search the current directory, then search the directory (or directories) in which system header files reside.
- The places to be searched for header files can usually be altered, often by a command-line option such as `-I`*path*.
- Don't use brackets when including header files that you have written:

```
#include <myheader.h>    /*** WRONG ***/
```

The **`#include`** Directive (cont.)

- The file name in an `#include` directive may include information that helps locate the file, such as a directory path or drive specifier:

```
#include "c:\cprogs\utils.h"
  /* Windows path */

#include "/cprogs/utils.h"
  /* UNIX path */
```

The **`#include`** Directive (cont.)

- 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"
```

The **#include** Directive (cont.)

- The `#include` directive has a third form:

  `#include` *tokens*

  tokens is any sequence of preprocessing tokens.

- The preprocessor will scan the tokens and replace any macros that it finds.

```
#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
```

Sharing Macro Definitions and Type Definitions

- Most large programs contain macro definitions and type definitions that need to be shared by several source files.
- These definitions should go into header files.
- For example, we can create a header file `boolean.h` to define two macros `TRUE` and `FALSE`, and one type `Bool`;

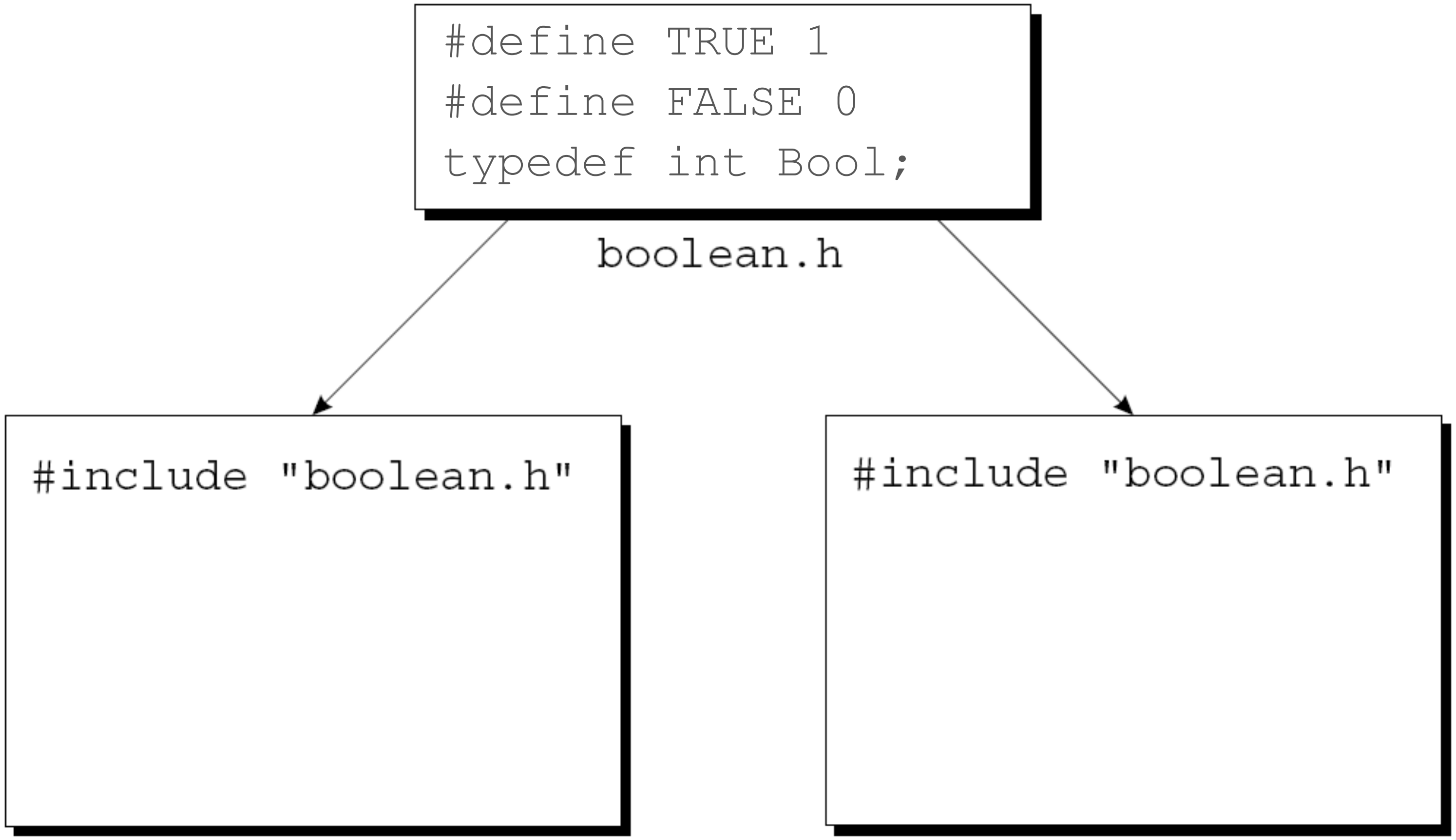
```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

- Any source file that requires the type and the macros will simply contain the line

```
#include "boolean.h"
```

Sharing Macro Definitions and Type Definitions (cont.)

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



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

```
#include "boolean.h"
```

```
#include "boolean.h"
```

Sharing Macro Definitions and Type Definitions (cont.)

- Advantages of putting definitions of macros and types in header files:
  - Saves time. We don't have to copy the definitions into the source files where they're needed.
  - Makes the program easier to modify. Changing the definition of a macro or type requires editing a single header file.
  - Avoids inconsistencies caused by source files containing different definitions of the same macro or type.

Sharing Function Prototypes

- Suppose that a source file contains a call of a function `f` that's defined in another file, `foo.c`.
- A better solution is to put `f`'s prototype in a header file (`foo.h`), then include the header file in all the places where `f` is called.
- We'll also need to include `foo.h` in `foo.c`, enabling the compiler to check that `f`'s prototype in `foo.h` matches its definition in `foo.c`.

Sharing Function Prototypes (cont.)

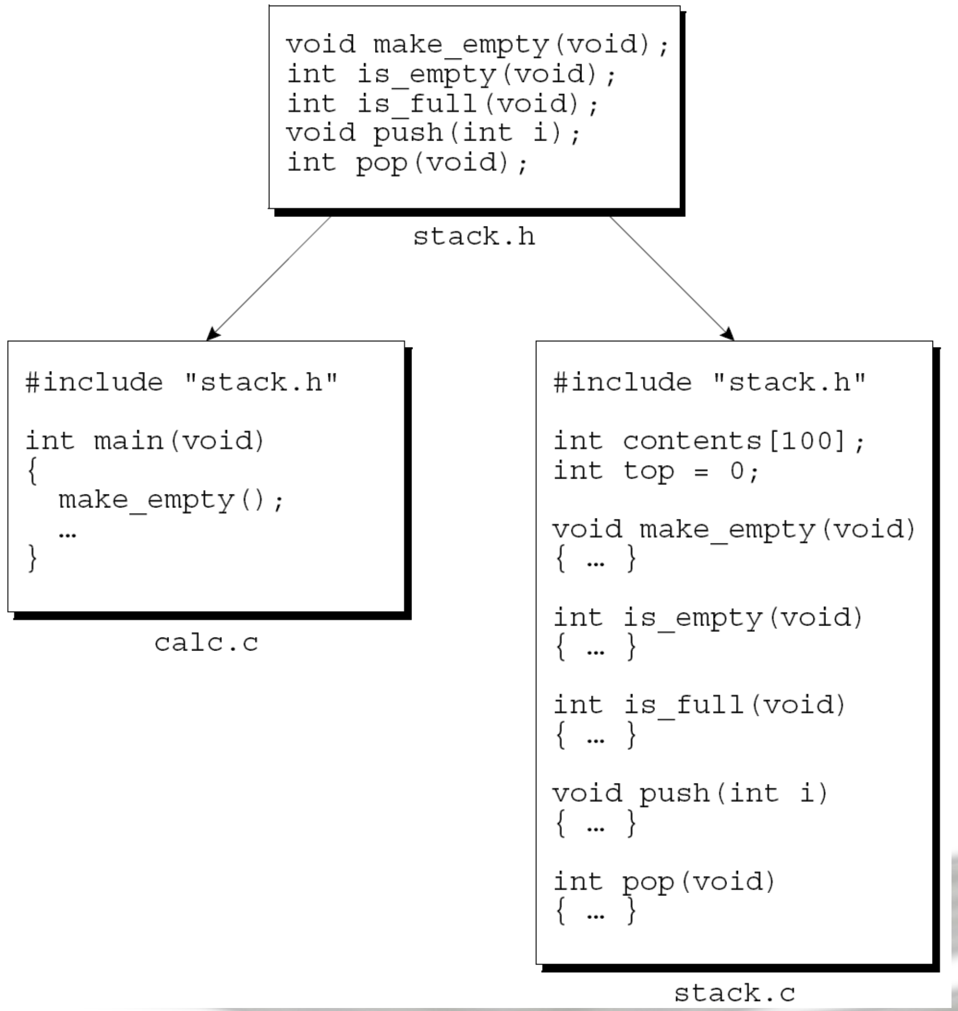
- The RPN calculator example can be used to illustrate the use of function prototypes in header files.
- The `stack.c` file will contain definitions of the `make_empty`, `is_empty`, `is_full`, `push`, and `pop` functions.
- Prototypes for these functions should go in the `stack.h` header file:

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

Sharing Function Prototypes (cont.)

- We’ll include `stack.h` in `calc.c` to allow the compiler to check any calls of stack functions that appear in the latter file.
- We’ll also include `stack.h` in `stack.c` so the compiler can verify that the prototypes in `stack.h` match the definitions in `stack.c`.

Sharing Function Prototypes (cont.)


stack.h

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

calc.c

```
#include "stack.h"

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

stack.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)
{ … }
```

Sharing Variable Declarations

- To share a function among files, we put its *definition* in one source file, then put *declarations* in other files that need to call the function.
- Sharing an external variable is done in much the same way.

Sharing Variable Declarations (cont.)

- An example that both declares and defines `i` (causing the compiler to set aside space):

```
int i;
```

- The keyword `extern` is used to declare a variable without defining it:

```
extern int i;
```

- `extern` informs the compiler that `i` is defined elsewhere in the program, so there's no need to allocate space for it.

Sharing Variable Declarations (cont.)

- When we use `extern` in the declaration of an array, we can omit the length of the array:

```
extern int a[];
```

- Since the compiler doesn't allocate space for `a` at this time, there's no need for it to know `a`'s length.

Sharing Variable Declarations (cont.)

- To share a variable `i` among several source files, we first put a definition of `i` in one file:

  ```
  int i;
  ```

- If `i` needs to be initialized, the initializer would go here.
- The other files will contain declarations of `i`:

  ```
  extern int i;
  ```

- By declaring `i` in each file, it becomes possible to access and/or modify `i` within those files.

Sharing Variable Declarations (cont.)

- When declarations of the same variable appear in different files, the compiler can’t check that the declarations match the variable’s definition.
- For example, one file may contain the definition

  ```
  int i;
  ```

  while another file contains the declaration

  ```
  extern long i;
  ```
- An error of this kind can cause the program to behave unpredictably.

Sharing Variable Declarations (cont.)

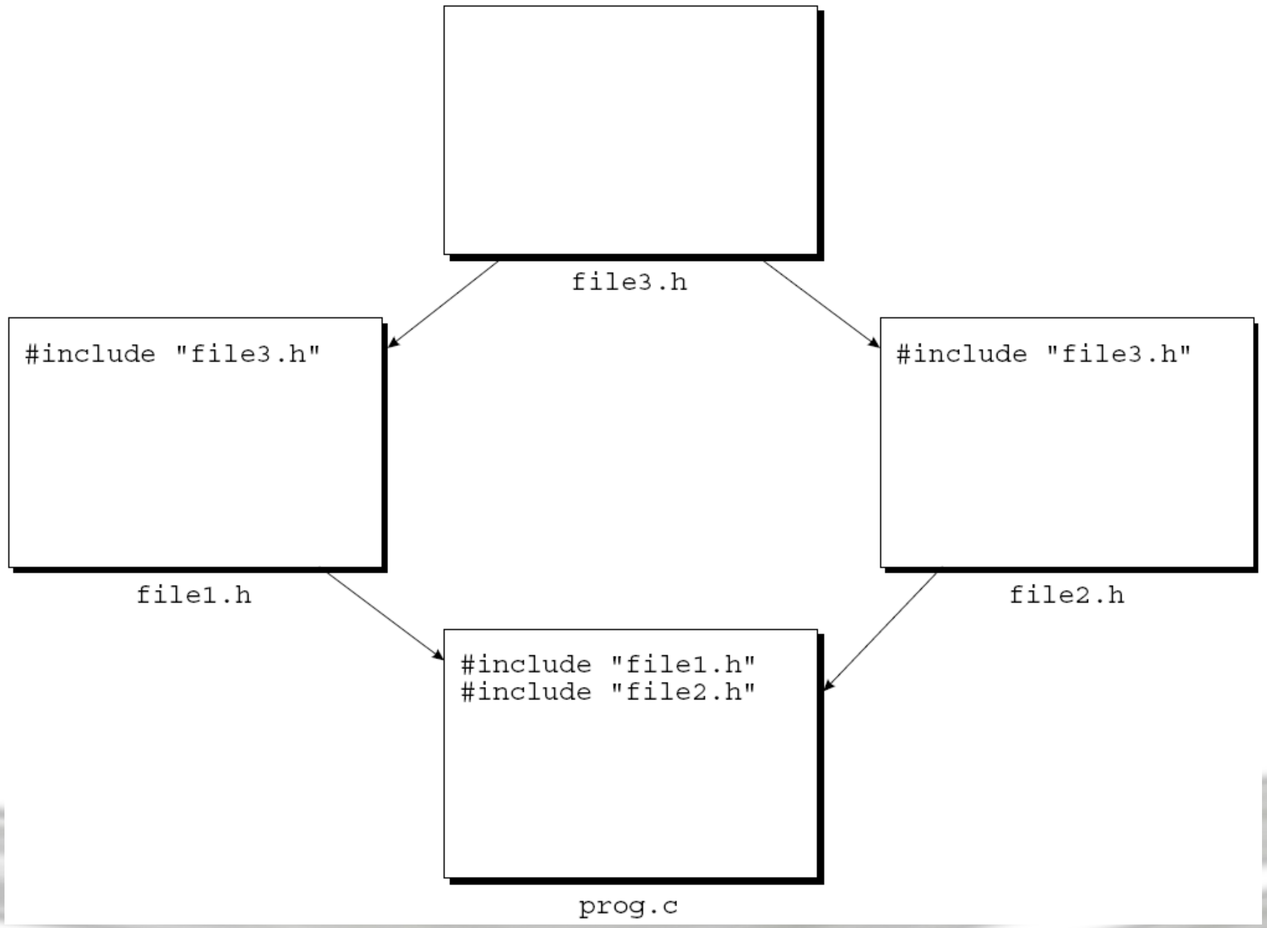
- To avoid inconsistency, declarations of shared variables are usually put in header files.
- A source file that needs access to a particular variable can then include the appropriate header file.
- In addition, each header file that contains a variable declaration is included in the source file that contains the variable’s definition, enabling the compiler to check that the two match.

Protecting Header Files

- If a source file includes the same header file twice, compilation errors may result.
- This problem is common when header files include other header files.
- Suppose that `file1.h` includes `file3.h`, `file2.h` includes `file3.h`, and `prog.c` includes both `file1.h` and `file2.h`.

Protecting Header Files (cont.)

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



Protecting Header Files (cont.)

- Including the same header file twice doesn’t always cause a compilation error.
- If the file contains only macro definitions, function prototypes, and/or variable declarations, there won’t be any difficulty.
- If the file contains a type definition, however, we’ll get a compilation error.

Protecting Header Files (cont.)

- Just to be safe, it's probably a good idea to protect all header files against multiple inclusion.
- 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
```

Since we can't name the macro `BOOLEAN.H`, a name such as `BOOLEAN_H` is a good alternative.

15.2 Dividing a Program into Files

Dividing a Program into Files

- Each set of functions will go into a separate source file (`foo.c`).
- Each source file will have a matching header file (`foo.h`).
  - `foo.h` will contain prototypes for the functions defined in `foo.c`.
  - Functions to be used only within `foo.c` should not be declared in `foo.h`.
- `foo.h` will be included in each source file that needs to call a function defined in `foo.c`.
- `foo.h` will also be included in `foo.c` so the compiler can check that the prototypes in `foo.h` match the definitions in `foo.c`.

Dividing a Program into Files (cont.)

- The `main` function will go in a file whose name matches the name of the program.
- It's possible that there are other functions in the same file as `main`, so long as they're not called from other files in the program.

Program: Text Formatting

- Let’s apply this technique to a small text-formatting program named `justify`.
- Assume that a file named `quote` contains the following sample input:

```
   C     is quirky,  flawed,    and  an
enormous   success.      Although accidents of   history
 surely  helped,   it evidently    satisfied   a   need

    for  a   system  implementation    language    efficient
 enough   to  displace         assembly   language,
   yet sufficiently   abstract   and fluent    to describe
  algorithms   and     interactions    in a   wide   variety
of   environments.
                     --      Dennis     M.        Ritchie
```

Program: Text Formatting (cont.)

- To run the program from a UNIX or Windows prompt, we'd enter the command

  ```
  justify <quote
  ```

- The `<` symbol informs the operating system that justify will read from the file `quote` instead of accepting input from the keyboard.
- This feature, supported by UNIX, Windows, and other operating systems, is called ***input redirection.***

Program: Text Formatting (cont.)

- Output of `justify`:

```
C is quirky,  flawed,  and  an  enormous  success.  Although
accidents of history surely helped, it evidently satisfied a
need for a system implementation language  efficient  enough
to displace assembly language, yet sufficiently abstract and
fluent to describe algorithms and  interactions  in  a  wide
variety of environments. -- Dennis M. Ritchie
```

- The output of `justify` will normally appear on the screen, but we can save it in a file by using ***output redirection:***

```
justify <quote >newquote
```

Program: Text Formatting (cont.)

- `justify` will delete extra spaces and blank lines as well as filling and justifying lines.
  - "Filling" a line means adding words until one more word would cause the line to overflow.
  - "Justifying" a line means adding extra spaces between words so that each line has exactly the same length (60 characters).
- Justification must be done so that the space between words in a line is equal (or nearly equal).
- The last line of the output won't be justified.

Program: Text Formatting (cont.)

- We assume that no word is longer than 20 characters, including any adjacent punctuation.
- If the program encounters a longer word, it must ignore all characters after the first 20, replacing them with a single asterisk.
- For example, the word

  ```
  antidisestablishmentarianism
  ```

  would be printed as

  ```
  antidisestablishment*
  ```

Program: Text Formatting (cont.)

- The program have to store words in a "line buffer" until there are enough to fill a line.
- The heart of the program will be a loop:

```
for (;;) {
  read word;
  if (can't read word) {
     write contents of line buffer without justification;
     terminate program;
  }
  if (word doesn't fit in line buffer) {
    write contents of line buffer with justification;
    clear line buffer;
  }
  add word to line buffer;
}
```

1. `flush_line()`
2. `space_remaining()`
3. `write_line()`
4. `clear_line()`
5. `add_word()`

Program: Text Formatting (cont.)

- The outline of the main loop reveals the need for functions that perform the following operations:
  1. Write contents of line buffer without justification
  2. Determine how many characters are left in line buffer
  3. Write contents of line buffer with justification
  4. Clear line buffer
  5. Add word to line buffer
- We'll call these functions `flush_line`, `space_remaining`, `write_line`, `clear_line`, and `add_word`.

Program: Text Formatting (cont.)

- The program will be split into three source files:
  - `word.c`: functions related to words
  - `line.c`: functions related to the line buffer
  - `justify.c`: contains the `main` function
- We’ll also need two header files:
  - `word.h`: prototypes for the functions in `word.c`
  - `line.h`: prototypes for the functions in `line.c`
- .

word.h

```c
#ifndef WORD_H
#define WORD_H

/**********************************************************
 * read_word: Reads the next word from the input and      *
 *            stores it in word. Makes word empty if no    *
 *            word could be read because of end-of-file.  *
 *            Truncates the word if its length exceeds     *
 *            len.                                         *
 **********************************************************/
void read_word(char *word, int len);

#endif
```

line.h

```c
#ifndef LINE_H
#define LINE_H
/*************************************
 * flush_line: Writes the current line
 *             without justification.
 *             If the line is empty,
 *             does nothing.
 *************************************/
void flush_line(void);
/*************************************
 * space_remaining: Returns the number
 *                  of characters left
 *                  in the current line.
 *************************************/
int space_remaining(void);
/*************************************
 * write_line: Writes the current line
 *.            with justification.
 *************************************/
void write_line(void);

/*************************************
 * clear_line: Clears the current line.
 *************************************/
void clear_line(void);

/*************************************
 * add_word: Adds word to the end of
 *           the current line. If this
 *           is not the first word on
 *           the line, puts one space
 *            before word.
 *************************************/
void add_word(const char *word);
#endif
```

Program: Text Formatting (cont.)

- Before we write the `word.c` and `line.c` files, we can use the functions declared in `word.h` and `line.h` to write `justify.c`, the main program.
- Writing this file is mostly a matter of translating the original loop design into C.

justify.c

```c
/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line();
  for (;;) {
      read_word(word, MAX_WORD_LEN+1);
      word_len = strlen(word);
      if (word_len == 0) {
        flush_line();
        return 0;
      }
      if (word_len > MAX_WORD_LEN)
        word[MAX_WORD_LEN] = '*';
      if (word_len + 1 >
            space_remaining()) {
        write_line();
        clear_line();
      }
      add_word(word);
    }
}
```

Program: Text Formatting (cont.)

- `main` uses a trick to handle words that exceed 20 characters.
- When it calls `read_word`, `main` tells it to truncate any word that exceeds 21 characters.
- After `read_word` returns, `main` checks whether `word` contains a string that's longer than 20 characters.
- If so, the word must have been at least 21 characters long (before truncation), so `main` replaces its 21st character by an asterisk.

Program: Text Formatting (cont.)

- The `word.h` header file has a prototype for only one function, `read_word`.
- `read_word` is easier to write if we add a small “helper” function, `read_char`.
- `read_char`’s job is to read a single character and, if it’s a new-line character or tab, convert it to a space.
- Having `read_word` call `read_char` instead of `getchar` solves the problem of treating new-line characters and tabs as spaces.

word.c

```c
#include <stdio.h>
#include "word.h"

int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
    return ' ';
  return ch;
}
```

```c
void read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
    ;
  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }
  word[pos] = '\0';
}
```

Program: Text Formatting (cont.)

- `line.c` supplies definitions of the functions declared in `line.h`.
- `line.c` will also need variables to keep track of the state of the line buffer:
  - `line`: characters in the current line
  - `line_len`: number of characters in the current line
  - `num_words`: number of words in the current line

line.c

```
#include <stdio.h>
#include <string.h>
#include "line.h"
#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

void clear_line(void)
{
  line[0] = '\0';
  line_len = 0;
  num_words = 0;
}
```

```
void add_word(const char *word)
{
  if (num_words > 0) {
    line[line_len] = ' ';
    line[line_len+1] = '\0';
    line_len++;
  }
  strcat(line, word);
  line_len += strlen(word);
  num_words++;
}

int space_remaining(void)
{
  return MAX_LINE_LEN - line_len;
}
```

```c
void write_line(void)
{
  int extra_spaces, spaces_to_insert, i, j;

  extra_spaces = MAX_LINE_LEN - line_len;
  for (i = 0; i < line_len; i++) {
    if (line[i] != ' ')
      putchar(line[i]);
    else {
      spaces_to_insert = extra_spaces / (num_words - 1);
      for (j = 1; j <= spaces_to_insert + 1; j++)
        putchar(' ');
      extra_spaces -= spaces_to_insert;
      num_words--;
    }
  }
  putchar('\n');
}
```

```c
void flush_line(void)
{
  if (line_len > 0)
    puts(line);
}
```

15.3 Building a Multiple-File Program

Building a Multiple-File Program

- Each source file (`.c`) in the program must be compiled separately.
- Header files don’t need to be compiled. The contents of a header file are automatically compiled whenever a source file that includes it is compiled.
- For each source file, the compiler generates an ***object files***—have the extension `.o` in UNIX and `.obj` in Windows.

Building a Multiple-File Program (cont.)

- The linker combines the object files created in the previous step—along with code for library functions—to produce an executable file.
- Among other duties, the linker is responsible for resolving external references left behind by the compiler.
- An external reference occurs when a function in one file calls a function defined in another file or accesses a variable defined in another file.

Building a Multiple-File Program (cont.)

- Most compilers allow us to build a program in a single step.
- A GCC command that builds `justify`:

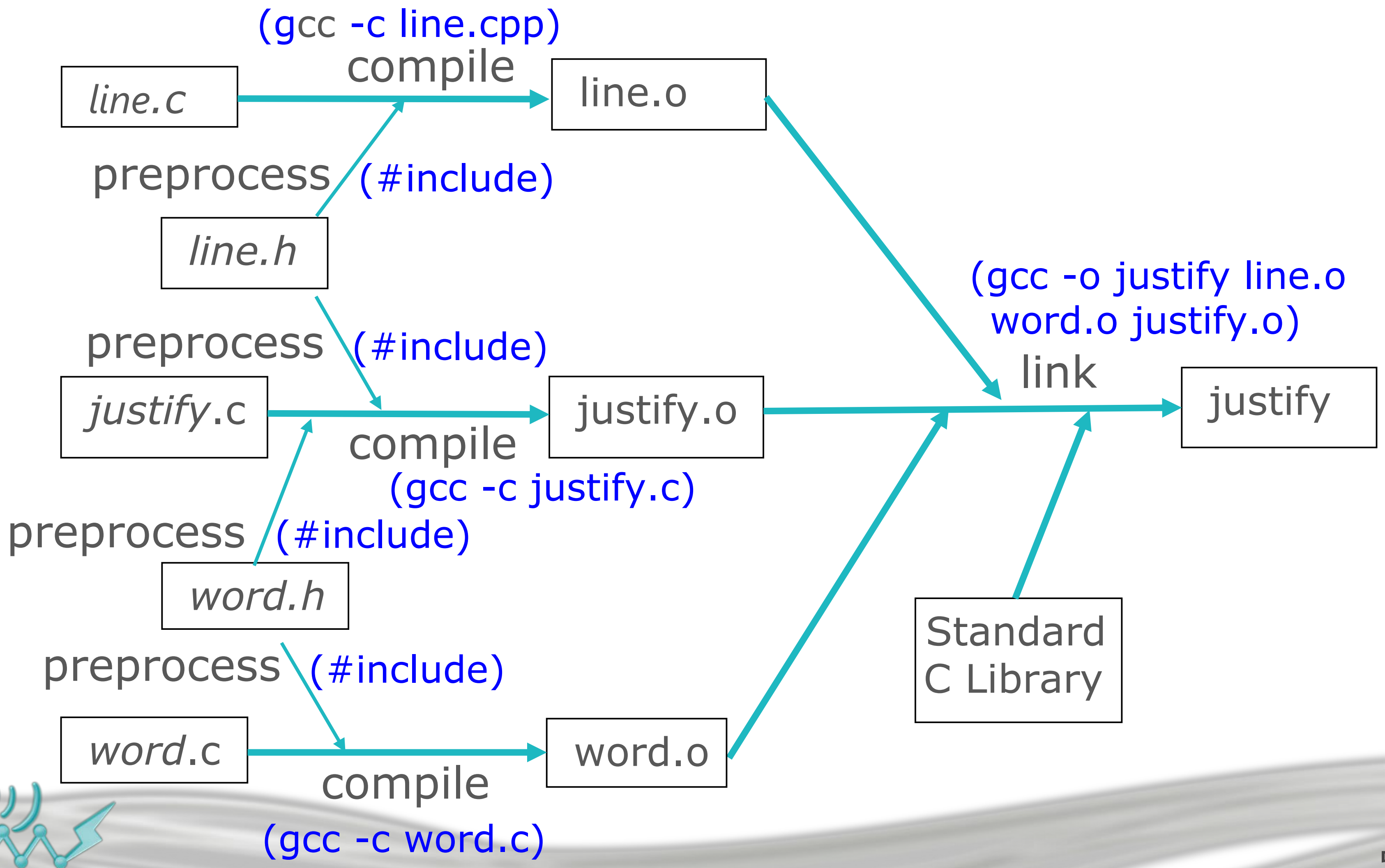
```
gcc -o justify justify.c line.c word.c
```

- 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`.

Makefiles

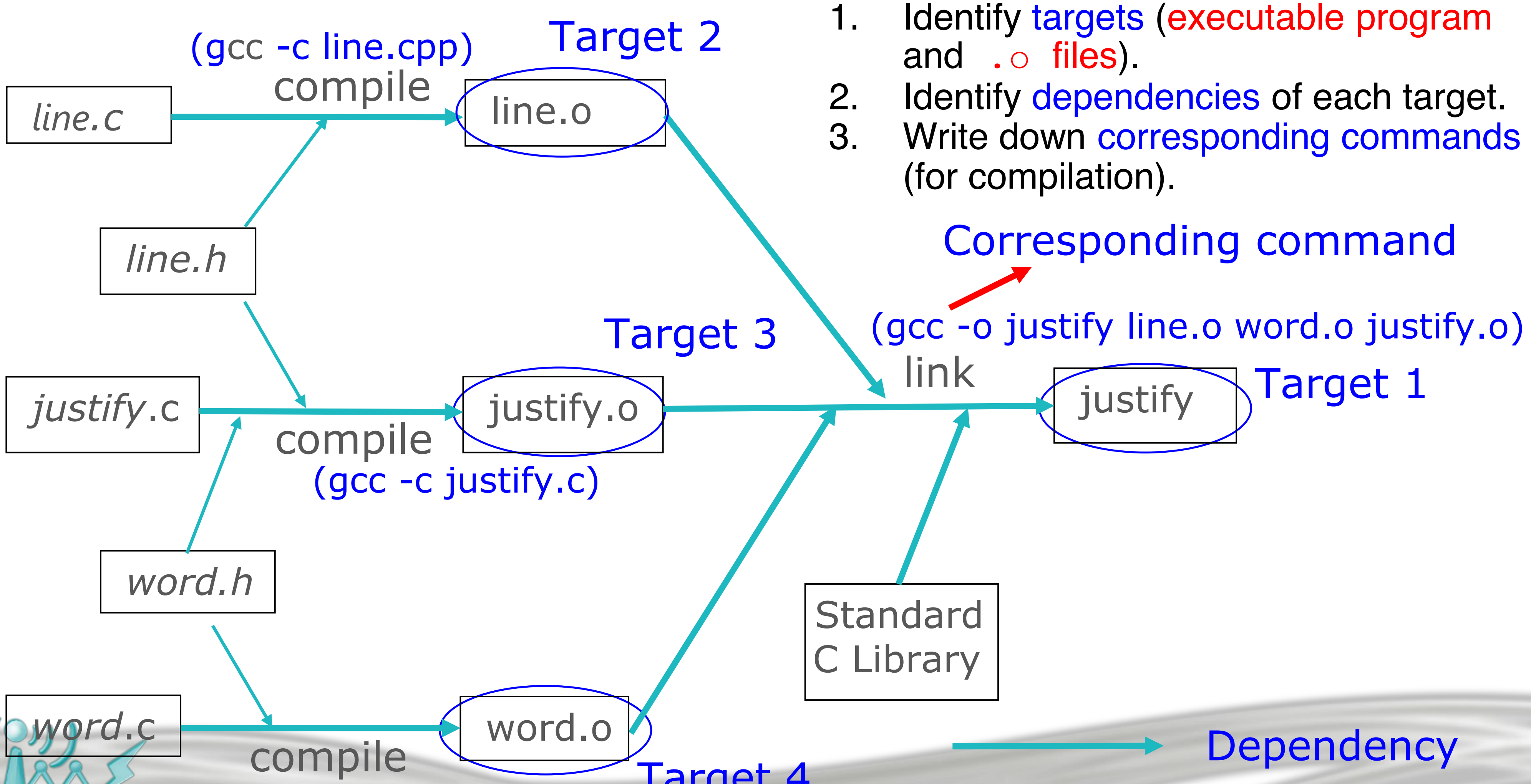
- To make it easier to build large programs, UNIX originated the concept of the ***makefile*.**
- A makefile not only lists the files that are part of the program, but also describes ***dependencies*** among the files.
- Suppose that the file `foo.c` includes the file `bar.h`.
- We say that `foo.c` "depends" on `bar.h`, because a change to `bar.h` will require us to recompile `foo.c`.

Makefiles (cont.)



Makefiles (cont.)

1. Identify targets (executable program and .o files).
2. Identify dependencies of each target.
3. Write down corresponding commands (for compilation).



Makefiles (cont.)

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

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

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

word.o: word.c word.h
    gcc –c word.c

clean:
    rm justify *.o
```

```
<target1>: <dependence>
<tab><command>
<tab><command>


<target2>: <dependence>
<tab><command>
```

Makefiles (cont.)

- There are four groups of lines; each group is known as a ***rule.***
- The first line in each rule gives a ***target*** file, followed by the files on which it depends.
- The second line is a ***command*** to be executed if the target should need to be rebuilt because of a change to one of its dependent files.

Makefiles (cont.)

- In the first rule, `justify` (the executable file) is the target:

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

- The first line states that `justify` depends on the files `justify.o`, `word.o`, and `line.o`.
- If any of these files have changed since the program was last built, `justify` needs to be rebuilt.
- The command on the following line shows how the rebuilding is to be done.

Makefiles (cont.)

- In the second rule, `justify.o` is the target:

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

- The first line indicates that `justify.o` needs to be rebuilt if there's been a change to `justify.c`, `word.h`, or `line.h`.
- The next line shows how to update `justify.o` (by recompiling `justify.c`).
- The `-c` option tells the compiler to compile `justify.c` but not attempt to link it.

Makefiles (cont.)

- Once we've created a makefile for a program, we can use the `make` utility to build (or rebuild) the program.
- By checking the time and date associated with each file in the program, `make` can determine which files are out of date.
- It then invokes the commands necessary to rebuild the program.

Makefiles (cont.)

- Each command in a makefile must be preceded by a tab character, not a series of spaces.
- A makefile is normally stored in a file named `Makefile` (or `makefile`).
- When the `make` utility is used, it automatically checks the current directory for a file with one of these names.

Makefiles (cont.)

- To invoke `make`, use the command

  `make` ***target***

  where *target* is one of the targets listed in the makefile.
- If no target is specified when `make` is invoked, it will build the first target.
- Except for this special property of the first rule, the order of rules in a makefile is arbitrary.

Errors During Linking

- If the definition of a function or variable is missing, the linker will be unable to resolve external references to it ("*undefined symbol*" or "*undefined reference*").
- Common causes of errors during linking:
  - ***Misspellings.*** If the name of a variable or function is misspelled, the linker will report it as missing.
  - ***Missing files.*** If the linker can't find the functions that are in file `foo.c`, it may not know about the file.
  - ***Missing libraries.*** The linker may not be able to find all library functions used in the program.
- In UNIX, the `-lm` option may need to be specified when a program that uses `<math.h>` is linked.

Rebuilding a Program

- To see how many files will need to be recompiled after a change, we need to consider two possibilities.
  - If we change a single source file, only that file must be recompiled.
  - If we change a header file, we should recompile all files that include the header file, since they are also potentially changed.
- By examining the date of each file, `make` can determine which files have changed since the program was last built.
- It then recompiles these files, together with all files that depend on them, either directly or indirectly.

Defining Macros Outside a Program

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

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

- In this example, the `DEBUG` macro is defined to have the value `1` for the program `foo.c`.
- If the `-D` option names a macro without specifying its value, the value is taken to be `1`.
- Many compilers also support the `-U` option, which "undefines" a macro as if by using `#undef`.