Writing Large Programs (2)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Last Lecture

- The `#include` Directive
  - three forms
- Sharing Macro Definitions and Type Definitions
- Sharing Function Prototypes
- Nested Includes
- Protecting Header Files

More details about Reverse Polish Notation

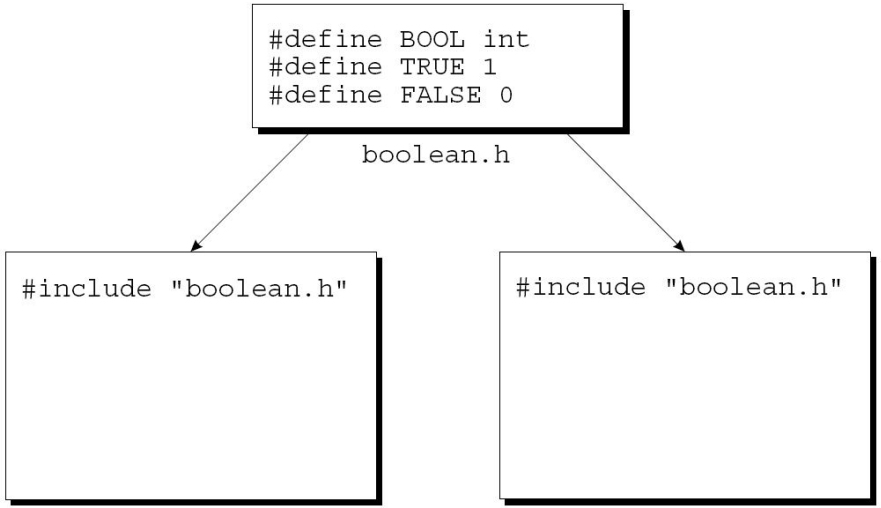
- Last week, we introduce RPN and computed `30 5 - 7 * = 175`
  - Why we subtract 5 from 30 (30 - 5) not 30 from 5 (5 - 30)?
- Short answer: the operands (values) will be computed following their order in the formula
  - For example, `3 4 5 × -`
    - `3 20 -`
    - `-17`
- extra reading: https://en.wikipedia.org/wiki/Reverse_Polish_notation

Outline

- Sharing Variable Declarations
- Dividing a Program into Files
- Building a Multiple-File program

Sharing Variable Declarations

- Like what we saw previously, 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

- In previous examples, we don’t need to distinguish between a variable’s declaration and its definition.
- The following statements declared and defined the varaibles at the same time

```
// declare i and c and define them as well
int i;
char c;
```

Sharing Variable Declarations

- To share external variables with different files, we need to separate the declaration and definition of the variables. How?
- The keyword `extern` is used to declare a variable without defining it
- `extern` informs the compiler that `i,c` are defined elsewhere in the program (mostly in a different source file), so there's no need to allocate space for them.

```
// declare i and c without defining it
extern int i;
extern char c;
```

Sharing Variable Declarations

- `extern` works with variable with all types!
- When we use `extern` in the declaration of an array, we can omit the length of the array:
- Since the compiler doesn't allocate space for `a` at this time, there's no need for it to know `a`'s length.

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

Sharing Variable Declarations

- To share a variable `i` among several source files, we first put a definition of `i` in one file
- If `i` needs to be initialized, the initializer would go here.

source_i.c

```
int i = 0;
```

Sharing Variable Declarations

- The other files will contain declarations of `i`
- By declaring `i` in each file, it becomes possible to access and/or modify `i` within those files.

source_i.c

```
#include "source_i.h"
int i = 100;
```

source_i.h

```
extern int i;
```

main.c

```
#include "source_i.h"
…
printf("%d", i);
…
```

Sharing Variable Declarations

- 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

- 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.

source_i.c

```
#include "source_i.h"
int i = 100;
char c = 'c';
```

source_i.h

```
extern int i;
extern char c;
```

main.c

```
#include "source_i.h"
…
printf("%d", i);
printf("%c", c);
…
```

Sharing Variable Declarations

- 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.

definition

source_i.c

```
#include "source_i.h"
int i = 100;
char c = 'c';
```

declaration

source_i.h

```
extern int i;
extern char c;
```

main.c

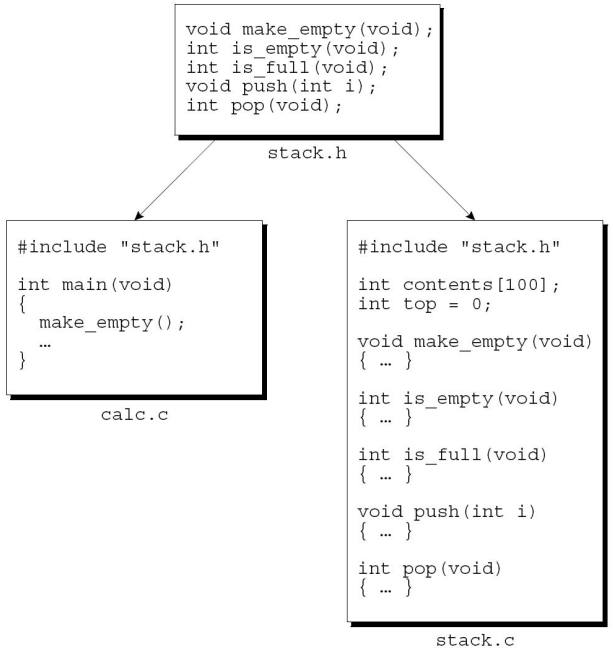
```
#include "source_i.h"
…
printf("%d", i);
…
```

Dividing a Program into Files

- After knowing how to use header files, `#include`, and source file to share macro, type definition, and function prototypes
- Let’s explore how to develop a program with multiple files by an example!

Dividing a Program into Files

- Remember!
- Each source file will have a matching header file and source file need include the header file
  - `stack.h` and `stack.c`
- The header file will be included in each source file that needs to call a function defined in the matching source file.
  - `stack.h` and `calc.c`
- The `main` function will go in a file whose name matches the name of the program.
  - `calc.c`



Example: Text Formatting

- Let’s apply this technique to a small text-formatting program.
- In this example, you will learn
  - Simple usage of how to use input and output **redirection** to read and output files in a C program
  - How to include header files and share function prototypes

Example: Text Formatting

- We will focus on how to connect each source file and header files, so we will not fully explain implementation details.
- We will provide the program files of this example on eCourse later.
  - Ch 15.3 provides detailed explanation of the implementations. Please check them by yourself!
  - Most of them are about character and string handling which we learned previously

Example: Text Formatting

- The text-formatting program is named `justify`
- Assume that a file named `input.txt` 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.

```

Example: Text Formatting

- After input the `input.txt` to `justify`, the output will look like this:
- `justify` will delete extra spaces and blank lines and filling and justifying lines.
  - adding words until one more word would cause the line to overflow.
  - adding extra spaces between words so that each line has exactly the same length (60 characters).

```
1  C is quirky, flawed, and an enormous success. Although
2  accidents of history surely helped, it evidently satisfied a
3  need for a system implementation language efficient enough
4  to displace assembly language, yet sufficiently abstract and
5  fluent to describe algorithms and interactions in a wide
6  variety of environments. -- Dennis M.
7
```

Example: Text Formatting

- To run the program from a UNIX or Windows prompt, we'd enter the command
  - `justify` is the exectuable file of this program
- The < symbol informs the operating system that justify will read from the file `input.txt` instead of **accepting input from the keyboard**.
- This feature, supported by UNIX, Windows, and other operating systems, is called ***input redirection.***

```
./justify <input.txt
```

Example: Text Formatting

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

```
1  C is quirky, flawed, and an enormous success. Although
2  accidents of history surely helped, it evidently satisfied a
3  need for a system implementation language efficient enough
4  to displace assembly language, yet sufficiently abstract and
5  fluent to describe algorithms and interactions in a wide
6  variety of environments. -- Dennis M.
7
```

```
./justify <input.txt >output.txt
```

Example: Text Formatting

- 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*`

Example: Text Formatting

- The program can’t write words one by one as they’re read.
- Instead, it will have to store them in a “line buffer” until there are enough to fill a line.

```
1  C is quirky, flawed, and an enormous success. Although
2  accidents of history surely helped, it evidently satisfied a
3  need for a system implementation language efficient enough
4  to displace assembly language, yet sufficiently abstract and
5  fluent to describe algorithms and interactions in a wide
6  variety of environments. -- Dennis M.
7
```

Program: Text Formatting

- The heart of the program will be a **loop** (modified from the example in the textbook)

```
while (1) {
        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;
}
```

Program: Text Formatting

- We can observe that the program need to deal with words and lines most of the time

```
while (1) {
        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;
}
```

Program: Text Formatting

- 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`

Building a Multiple-File Program

- Each source file in the program must be compiled.
- 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 a object file containing object code.
  - For example, files with extension `.o` in UNIX

Building a Multiple-File Program

- 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
```

Building a Multiple-File Program

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

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

Summary

- Sharing Variable Declarations
  - `extern int i;`
- Dividing a Program into Files
  - Example: Text Formatting
  - Activity
- Building a Multiple-File program
  - command line: `gcc -o justify justify.c line.c word.c`