

CS113: Lecture 7

Topics:

- The C Preprocessor
- I/O, Streams, Files

Remember the name: Pre-processor

- Most commonly used features: `#include`, `#define`.
- Think of the preprocessor as processing the file so as to “remove” all `#whatevers`.
- `#include` inserts the entire contents of a file, verbatim, exactly where the `#include` appeared.

Two ways to use:

- `#include "filename"` - typically searches for `filename` in the directory where the source program is
- `#include <filename>` - searches for `filename` in an implementation-defined way.

Typically, use angle brackets for standard library header files.

#define: a macro facility

```
#define SIZE 10
#define CENT_TO_INCHES 2.54

void main() {
    int i;
    int inches[SIZE], centimeters[SIZE];

    /* read in inches[] ... */

    for( i = 0; i < SIZE; i++ )
        centimeters[i] = inches[i] * CENT_TO_INCHES;
}
```

Why use?

- Making global changes is easier.
- Makes programs easier to read (assuming descriptive names)

Some will go so far as to say that no constants should appear in your code (other than perhaps some zeroes and ones) - they should all be `#defined`.

#define: more details

- Possible to define macros with arguments:

```
#define sum(A,B) ((A)+(B))
```

```
x = sum( 3 * x, 1 ); becomes
```

```
x = (( 3 * x ) + ( 1 ));
```

- Put brackets around anything that could be a more complex expression!
- For example, this doesn't work:

```
#define square(x) x * x /* Doh! */
```

```
y = 2 * square(z + 1);
```

- One use of #define is to speed up programs by avoiding the overhead of function calls. Another use is to improve readability and make programs “self-documenting”:

```
#define max(x,y) ((x) > (y) ? (x) : (y))
```

More on the preprocessor

- Conditional compilation:

```
#define DEBUG

void main() {
    int x;

    /* blah blah blah */

    #ifdef DEBUG
        printf( "current value of x is %d\n", x );
    #endif
}
```

- There are other features: `#if`, `#else`, etc.

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
```

- Use conditional compilation for testing! With a single file you can have “testing” and “ship” versions of your program, where in the testing code you can include lots of tests to check for null pointers, etc., that would be unnecessary and slow in the ship version. (Such tests are often called “assertions”.)

Files

The library `stdio.h` defines a structure called `FILE`. (We'll learn more about structures next time.) You don't need to know any of the details of how a `FILE` works, you just need to know how to point to them. A pointer to a file is called a "file handle".

Here's the paradigm:

```
void main() {  
  
    FILE *fp; /* pointer to a file */  
  
    /* open WRITEME.txt for reading */  
    fp = fopen("WRITEME.txt", "w")  
  
    /* do stuff with the file */  
    fprintf(fp, "This goes in the file");  
  
    /* close the file -- it's a good habit */  
    fclose(fp);  
  
}
```

The function `fopen` accepts a string corresponding to the name of the filename, and another string that says whether we want to open the file to read ("`r`"), write ("`w`"), or append ("`a`").

- If there is any error, `fopen` returns the pointer value `NULL`.

Printf and Scanf with Files

Up to now, we've been using `printf` to print messages (to "standard output"), and `scanf` to receive input from the keyboard (from "standard input").

These can be generalized to the functions `fprintf` and `fscanf`, which write to and read from a *particular file*:

- Given a file pointer `fp`, the statement:
`fprintf(fp, "This goes in the file");`
writes to the file handled by `fp`.
- The statement:
`fscanf(fp, "%d", &intvar);`
reads an integer from the file.

C treats the standard input and output as files

- Their file handles are defined as the constants `stdin` and `stdout` in `stdio.h`.
- So, `printf` and `scanf` are just special cases of `fprintf` and `fscanf`, applied to the file handles `stdout` and `stdin`.

```
printf("Hello, world!");  
/* is equivalent to */  
fprintf(stdout, "Hello, world!");
```

Other ways to read and write

There are more rudimentary ways to read from files and write to files. Suppose that `fp` is a file handle.

- The function `getc(fp)` returns a single character read from the file.
- If you want to put a character *back* in the stream, use the function `ungetc(c,fp)`.
- `putc(c,fp)` writes the character `c` to the file.
 - It returns the character `c` if successful, or the special character `EOF` (a constant defined in `stdio.h`, for “end of file”) if the write fails.

Here’s how to use the preprocessor to write new “functions” that read characters from standard input and write to standard output:

```
#define getchar()    getc(stdin)
#define putchar(c)  putc((c),stdout)
```

(Actually, `getchar` and `putchar` are already defined, and they work exactly as defined here.)

Reading/writing strings

We can read strings from files, too:

- `fgets(char_array,max_characters,fp)` copies at most `max_characters` characters from the file handled by `fp`.
- The standard usage is this:
`fgets(my_char_array,sizeof(my_char_array),fp)`
- The `sizeof` function returns the number of *bytes* in the structure you pass to it. Since a `char` takes up a single byte, you can use it to find the number of elements in a char array!

It's MUCH BETTER to use `fgets` than `scanf` to read strings from standard input. Why?

- If the string read by `scanf` is longer than the buffer allocated for it, `scanf` will crash, probably through some ugly memory error.
- If the string you type doesn't match the format string you use as an argument to `scanf`, the function will probably crash.
 - Like, if you're trying to read an integer and the user types a letter.

How fgets works

- Blocks the program and reads from the file (or standard input) until a newline character.
- The resulting string *includes* the newline character (unless it fills up the whole buffer).
 - How would you trim the newline character from the string?
- Stores up to `max_characters-1` characters in the buffer. (All strings are terminated by 0!)
- If more than that number of characters is encountered before a newline, they'll be in the stream the next time you read from that file (using `fgetc`, `fgets`, `fscanf`, etc.).

But what about the useful string parsing that `scanf` does?

Using sscanf and sprintf

- It turns out that `fscanf` and `fprintf` have versions that apply to *strings*.
- The function `sscanf` works exactly like `fscanf`, only instead of passing it a file handle, you pass it a string to read from.
- The function `sprintf` works the same – instead of passing it a file handle to write to, you pass it a string buffer (in this example, `s`) to write to.
 - `sprintf(s, format_string, arg1, arg2, ...)`
 - It doesn't *return* a string, like you might see in Java. Remember: you always have to allocate space for strings, and the assignment operator doesn't work for arrays.

The “safe” string-reading paradigm

To read a string from standard input:

```
char *string_buffer[100];
int a;

printf( "Enter a number: " );
fgets( string_buffer, sizeof(string_buffer), stdin );

/* do stuff with the string */
printf( "String length: %d\n", strlen(string_buffer) );
printf( "String entered: %s", string_buffer );

/* parse the integer */
sscanf( string_buffer, "%d", &a );

/* etc. */
```

This paradigm won't crash your program. Users may still enter bad data, but you can determine that and deal with it within the program, rather than finding out when the whole program crashes.

More on sscanf

The function `sscanf` returns an `int` equal to the number of tokens that were matched.

Example: using `sscanf` to parse a date string

```
if( sscanf( buffer, "%d %s %d", &day,
           monthname, &year ) == 3 ) {
    /* 25 Dec 1988 form */
    printf( "valid: %s\n", buffer );
} else if( sscanf( buffer, "%d/%d/%d",
                  &month, &day, &year ) == 3 ) {
    /* mm/dd/yy form */
    printf( "valid: %s\n", buffer );
} else {
    printf( "invalid: %s\n", buffer );
}
```

For more information, see Chapter 7 of K&R.