CS113: Lecture 10

Topics:

- I/O
- Style
I/O: sprinting

- We have been using the functions `printf` to print out messages, and `scanf` to receive input.

- `sprintf` is the same as `printf`, but accepts one additional parameter, a string (the first parameter). The output is placed in this string instead of being output to the screen.

Example:

```c
int a = 3, b = 5;
char s[80];

sprintf( s, "a is %d, b is %d\n", a, b );
```
I/O: scanf, sscanf

- `sscanf` is also the same as `scanf`, but accepts one additional parameter, a string; `sscanf` reads from this string instead of from the keyboard.

`scanf` and `sscanf` both return an int equal to the number of tokens that were matched.

Example:

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

- For more information, see Chapter 7 of K&R.
File I/O

- To write to/read from files, one needs to first call fopen to obtain a “file handle”, or file pointer which will be used to access the file. File pointers have type FILE *. After one is done using the file, fclose should be called.

- fopen accepts two parameters, the file name and the mode of access: "w" for writing, and "r" for reading.

- Example:

```c
FILE *in_file;
int a;

in_file = fopen( "input.txt", "r" );
if( in_file == NULL )
{
    printf( "Error opening file input.txt.\n" );
    exit( 1 );
}
scanf( in_file, "%d", &a );
fclose( in_file );
```

- Once the file has been opened, one can read/write using the functions fscanf and fprintf; these accept as their first parameter file pointers. All other parameters are interpreted as they would be in a call to scanf or printf.
Style: Flow control

In an if/else statement, write the shorter clause first:

```c
if( condition )
{
    a = b + c;
    initialize( &a );
    do_stuff();
    do_more_stuff();
}
else
{
    a = b - c;
}

becomes

if( !condition )
{
    a = b - c;
}
else
{
    a = b + c;
    initialize( &a );
    do_stuff();
    do_more_stuff();
}
```
Style: Flow control

Remember that else is unnecessary after return, break, and continue.

Before... and after...

```c
if( a < b )
{
    do_this();
    return;
}
else
{
    a = b + c;
    if( joe )
    {
        process( a );
    }
    else
    {
        process( b );
    }
    blah();
    blah_blah();
}
```

Removing such `else`es reduces the amount of indentation necessary.
Style: Flow control

Minimize nesting.

Too many levels of nesting make code less readable. To reduce the amount of nesting, use return, break, and continue, and negate conditions.

```c
if( string[1] == '!' )
{
    if( string[2] == '$' )
    {
        do_that();
        do_this();
    }
    else
    {
        return( yo );
    }
}
else
{
    return( joe );
}
```

becomes

```c
if( string[1] != '!' )
    return( joe );
if( string[2] != '$' )
    return( yo );
do_that();
do_this();
...
```
Curiosity: A self-reproducing program

Note that 34 is the ASCII value of the double-quote character.

```c
char*s="char*s=%c%s%c;main(){printf(s,34,s,34);}"
main(){printf(s,34,s,34);}
```

(There should be no carriage return in the middle of the program; I inserted one for the sake of formatting.)