Controlling the Command Line

Output redirection

The *fprintf(stdout,)* function sends its output to a logical file commonly known as the *standard output* or simply *stdout.*

When a program is run in the Unix environment, the logical file *stdout* is by default associated with the screen being viewed by the person who started the program.

The > operator may be used on the command line to cause the standard output to be *redirected* to a file:

```
acad/cs101/examples/notes ==> p8 > squares.txt
```

A file created in this way may be subsequently viewed using the *cat* command (or edited using a text editor).

```
class/215/examples ==> cat squares.txt | more

1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
```

Input redirection

Like the *stdout* the *stdin* may also be redirected.   To redirect both *stdin* and *stdout* use:

```
p4 <  p4in.txt  > p4out.txt
```

when invoked in this manner when the program *a.out* reads from the *stdin* via *scanf()* or *fscanf(stdin,.)* the input will actually be read from a file named *input.txt* and any data written to the *stdout* will end up in the file *output.txt.*

```
acad/cs101/examples/notes ==> cat p4in.txt
3 4
5 6
7 8
9 11
```

```
acad/cs101/examples/notes ==> p4 < p4in.txt > p4out.txt
```

```
acad/cs101/examples/notes ==> cat p4out.txt
3  +  4 =  7
5  +  6 =  11
7  +  8 =  15
9  +  11 =  20
```

Command line arguments

It is often useful to pass arguments to a program via the command line. For example,

```
gcc -g -Wall -o p10 p10.c
```

In this case the C compiler, *gcc,* is being passed 6 different arguments.

```
0   gcc
1   -g
2   -Wall
3   -o
4   p10
5   p10.c
```

Printing command line arguments

When a program is started from the command line, the character strings (separated by spaces) comprising the program name and the remaining arguments are copied by the Operating System into memory space occupied by the new program and a table or array of addresses is passed to the main function. These values can be accessed by the *main()* function as shown below.

```
/* printargs.c */

#include <stdio.h>

int main(
int argc,      /* number of command line arguments    */
char *argv[]) /* array of addresses of the arguments */
{
   int ndx = 0;

   while (ndx < argc)
   {
      fprintf(stdout, "%s\n", argv[ndx]);
      ndx = ndx + 1;
   }
}
```

When the program is invoked as follows:

```
==> printargs -Wall -o hello -g myprog.c

printargs
-Wall
-o
hello
-g
myprog.c
```

Processing numerical values

Suppose you mission is to write a program named *flag* whose function is to produce a .ppm image of a flag. Your program is to be invoked as:

```
    flag width-of-flags-in-pixels
    ---
    flag 800
```

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

int main(
int argc,    /* number of cmd line args */
int *argv[]) /* array of arg addresses  */
{
   int width = 0;
   int howmany = 0;

   if (argc < 2)
   {
      fprintf(stderr, "usage is flag width-in-pix\n");
      exit(1);
   }

   howmany = sscanf(argv[1], "%d", &width);

   fprintf(stderr, "Width = %d \n", width);
}
```