The Standard I/O Library

The C language itself does not define any particular file or character-based input or output routines (nor any windowing routines) – unlike Java and Pascal Instead any program may provide its own. Clearly this is a daunting task, and so the standard C library provides a collection of functions to perform file-based input and output. The standard I/O library functions provide efficient, *buffered* I/O to and from both terminals and files.

C programs requiring standard I/O should include the line:

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

All transactions through the standard I/O functions require a *file pointer*:

```
FILE  *fp;

fp = fopen("file.dat", "r");
 ......
fclose(fp);
```

Although we are strictly dealing with a C *pointer*, we simply pass this pointer to functions in the standard C library. Some texts will refer to this pointer as a *file stream* (and C++ confused this even more), but these should not be confused with nor be described as akin to Java's streams.

An number of predicate macros are provided to check the status of file operations on a given file pointer:

```
feof(fp)          /* checks for end-of-file */
ferror(fp)        /* checks for an error on a file */
```

The standard I/O functions all return `NULL` or `-1` (as appropriate) when an error is detected. For example:

```
#include  <stdio.h>

int main()
{
  FILE *fp;

  if((fp=fopen("/etc/passwd", "w")) == (FILE *)NULL) {
      error message ...
  }
  else {
   /* process the file */
       ...
      fclose(fp);
  }
}
```


The Standard I/O Library, *continued*

The most frequently used functions in the C standard I/O library perform output of formatted data. We also see here the most frequent use of C's acceptance of functions receiving a variable number of arguments:

```
fprintf(FILE *fp, char *format, (T)arg1, (T)arg2, ...);

e.g.   int  res;
       char *name = "Chris";

       fprintf(fp,"res=%d name=%20s\n", res, name);
```

Many standard I/O functions accept a *format specifier* – a string indicating how following arguments are to be displayed. This mechanism is in contrast to Java's `toString` facility in which each object knows how to output/display itself as a `String` object. There are many possible format specifiers, the most common ones being 'c' for character values, 'd' for decimal values, 'f' for floating point values, and 's' for character strings. Format specifiers may be preceded by a number of format modifiers, which may further specify their data type, and to indicate the width of the required output (in characters).

As a special case, we may use a more concise version of `fprintf()` in which the `FILE` pointer of the operating system's standard output device is used (typically, the screen). Thus, the following two statements are identical:

```
fprintf(stdout, "res=%d name=%20s\n", res, name);
printf("res=%d name=%20s\n", res, name);
```

We mentioned before that the C standard I/O library provides efficient buffering. This means that although it appears that the output has "gone" to the `FILE` pointer, it may still be held within an internal character buffer in the library (and will hence not yet be on disk, or to the screen). We often need to *flush* our output to ensure that it is more quickly written to disk or the screen. `FILE` pointers are automatically flushed when a file is closed or the process exits:

```
/* ... format some output ...*/
fflush(fp);
```

As well as outputting to `FILE` pointers, we may also perform formatted output to a character array (a string), with a very similar series of functions:

```
int  res;
char *name = "Chris";
char buffer[BUFSIZ];

sprintf(buffer, "res=%d name=%20s\n", res, name);
```

The Standard I/O Library, *continued*

C's standard I/O library may also be used to input values from `FILE` pointers and character arrays using `fscanf()` and `sscanf()`. Because we want the contents of C's variables to be modified by the standard I/O functions, we need to pass the *address* of the variables:

```
        fscanf(fp, format, &arg1, &arg2, ...);

e.g.    int i, res;
        char buffer[BUFSIZ];

        fscanf(fp, "%d %d", &i, &res);
        sscanf(buffer, "%d %d", &i, &res);
```

We also frequently need to read all lines from a file, or to (perhaps) sum all integers values from a file. We must be careful here, with the particular return values of the C standard I/O functions. The functions themselves return NULL `FILE` pointers, or a value of −1 at the end of a file or an error condition, but we must be care *when* we check these values:

```
#define  MAXLINE      80

int  i, sum;
char line[MAXLINE];

for(;;) {
    fgets(line, sizeof(line), fp);
    if(feof(fp))
        break;

    /* ... process the line just read ...*/
}
fclose(fp);

 ........

sum = 0;
while(fscanf(fp, "%d", &i) == 1)
    sum += i;
fclose(fp);
```


The C/Operating System Interface

Operating systems, such as UNIX, LINUX and Windows-NT, will call C programs with three parameters:

- an **int**eger argument count (`argc`),
- an array of pointers to character strings (`argv`), and
- an array of pointers to character strings (`envp`).

Notice that in many previous examples we've provided a `main()` without any parameters all. Remember that C does not check the length and types of parameter lists of functions which it does "not know" about – ones that have not been prototyped. In addition, the function `main()` has no special significance to the C compiler. Only the linker requires `main()` as the apparent starting point of any program. Most C programs you see will only have the first two parameters.

```
int main(int argc, char *argv[], char *envp[])

-> explain char *argv[]
   declare argv as array of pointer to char
```

The C/Operating System Interface, *continued*

A common activity at the start of a c program is to search the argument list for command-line switches commencing with a '–' character. Remaining command-line parameters are often assumed to be filenames:

```
int main(int argc, char **argv)
{
    argv0 = (argv0 = strrchr(argv[0],'/')) ? argv0+1 : argv[0];
    argc--; argv++;
    while((argc > 0) && (*argv[0] == '-')) {
        switch (*argv[1]) {
            case 'd' :  dflag = !dflag;
                        break;
            default :   argc = 0;
                        break;
        }
        argc--; argv++;
    }
    if(argc < 0) {
        fprintf(stderr, "Usage : %s %s\n",argv0,usage);
        exit(1);
    }
    if(argc > 0)
        while(argc > 0) {
            process(*argv);
            argv++; argc--;
        }
    else
        process((char *)NULL);
    return(0);
}
```


Dynamic Memory Allocation

The function `malloc()` returns a requested number of bytes from the operating system's heap. If insufficient memory is available `malloc` returns `NULL`. When we are finished using the space returned by `malloc()`, our program should be returned to the heap with a call to `free()`. If a process continues to `malloc()` memory and fails to deallocate it using `free()`, the process will quickly "run out of memory" and terminate ungracefully.

Unlike Java, C has no garbage collection of heap objects, and so programs must be very careful about deallocating memory that is no longer required.

Consider the following example which allocates space for a new copy of a given string. This is very similar to the standard function named `strdup()`:

```
char *newstr(const char *s)
{
    void *malloc(unsigned int nbytes);
    char *p;

    if( (p=(char *)malloc(strlen(s)+1)) == NULL ) {
        fprintf(stderr,"out of memory!\n");
        exit(1);
    }

    strcpy(p,s);
    return(p);
}
```

`malloc()` is also frequently used to allocate memory for structures.

```
#define  NEW(t)      (t *)malloc(sizeof(t))

struct l {
     char *line;
     struct l *next;
};

struct l *hd = (struct l *) malloc( sizeof(struct l) );

fgets(buf, MAX, fp);
while( !feof(fp) ) {
    p       = NEW(struct l);
    p->line = newstr(buf);
    p->next = hd;
    hd      = p;
    fgets(buf,MAX,fp);
}
```

C Projects Developed In Multiple Files

Just as programs should be divided into functions and procedures, C programs themselves should be partitioned into files. Then, small changes to one file do not require all files to be recompiled.

Assume that a project is partitioned into the files

```
proj.c  data.c  io.c  and  calc.c
```

Each file depends on a common header file which we will name `proj.h`.
In the header file :

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

#define  FALSE       0
#define  TRUE        1

#define  MAX        80
#define  NEW(t)     (t *)malloc(sizeof(t))

struct l {
    char *l;
    struct l *next;
} ;

extern struct l *hd;

extern char buf[];

extern int linecount;
```

In the file data.c :

```
#include  "proj.h"

struct l  *hd;

char buf[MAX];

int linecount;
```


C Projects Developed In Multiple Files, *continued*

In other files :

```
#include  "proj.h"

int main(int argc, char *argv[])
{
     .... process arguments

    openfiles();        /* in  io.c   */
    doit();             /* in  calc.c  */
    exit(0);
}
```

Compiling Multiple Files Using *make*

The program *make* maintains up-to-date versions of programs that result from a sequence of operations on a set of files.

make reads specifications from either *makefile* or *Makefile* and performs the actions associated with *rules* if indicated files are “out of date”. Basically:

```
if (files on which a certain file depends)
       i) do not exist, or
      ii) are not up-to-date
then
     create an up-to-date version;
```

make operates over rules and operations recursively and will abort if it cannot create an up-to-date file on which another file depends.

Compiling Multiple Files, *continued*

For the case of our multi-file project earlier :

```
proj : proj.o data.o io.o calc.o
       gcc -o proj proj.o data.o io.o calc.o


proj.o : proj.c proj.h
       gcc -c proj.o proj.c

data.o : data.c proj.h
       gcc -c data.o data.c

io.o : io.c proj.h
       gcc -c io.o io.c

calc.o : calc.c proj.h
       gcc -c calc.o calc.c
```

make also supports simple variable and macro substitutions :

```
NAME   = proj
OBJ    = proj.o data.o io.o calc.o

CC     = gcc
CFLAGS = -O


$(NAME) : $(OBJ)
       $(CC) -o $(NAME) $(OBJ)


proj.o : proj.c proj.h
       $(CC) $(CFLAGS) -c proj.o proj.c

data.o : data.c proj.h
       $(CC) $(CFLAGS) -c data.o data.c

io.o : io.c proj.h
       $(CC) $(CFLAGS) -c io.o io.c

calc.o : calc.c proj.h
       $(CC) $(CFLAGS) -c calc.o calc.c


clean :
       rm -f $(OBJ) core
```