Lecture 7. Standard I/O Library

Formatted I/O

Formatted output is handled by the three `printf()` functions:

```
#include <stdio.h>

int printf(const char *format, …);

int fprintf(FILE *fp, const char *format, …);
```
Both return number of characters output if OK, negative value if error.

```
int sprintf(char *buf, const char *format, …);
```
Returns number of characters stored in array.

Arguments:

- *format* — specifies the output format for numeric values. Each format specification contains the following characters, in order:

    Flags:
    - "%" — the percent character precedes all formats
    - "#" — certain format specifiers will cause the output to be printed in alternate form:
        - for `o` (octal), the leading `0` will be printed;
        - for `x` (hexadecimal), the leading `0x` will be printed;
        - for `e, E, f, F, g, G` (floating point formats), the decimal point will be printed regardless;
    - "-" — output will be left-justified in the specified output field;
    - "+" — output will be right-justified;
    - " " — a blank should be used to the left of a positive number, to allow for a minus sign for the negative numbers without losing alignment;
    - `0` — a blank padding (for instance, b/w a number and a unit of measure).

    Field width:
    - *digit* — specifying the width of the field, padded with blanks if needed.

    Precision:
    - *.digits* — specifying the number of decimals.

    Format:
    - *character* — specifying the format (see table)

- *fp* — pointer to a FILE structure
- *buf* — char pointer taking contents of the formatted output

Below is the table of format specifiers:

Format Character(s)	Meaning
di	decimal
u	unsigned decimal
o	octal
xX	hexadecimal
fF	`[-]ddd.ddd` with # `ds` specified
eE	`[-d.ddd+-dd]` (scientific notation)
gG	whichever of `fF` or `eE` gives full precision in minimum space
aA	`[-h.hhh+-pd]` (for hex #s, p precision)
c	print 1st character of output
s	string
b	interpret escaped characters in string
%	print a literal %

The next three variants of `printf()` have the … variable argument list replaced with an argument list *arg*:

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

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *fp, const char *format, va_list arg);
```

Both return number of characters output if OK, negative value if error.

```
int vsprintf(char *buf, const char *format, va_list arg);
```

Returns number of characters stored in array.

Functions with Variable Argument Lists

`va_list` is a data type defined in `<stdarg.h>`:

```
typedef struct
{
     unsigned int gp_offset;
     unsigned int fp_offset;
     void *overflow_arg_area;
     void *reg_save_area;
} va_list[1];
```

The following macros (functions) can be used with a `va_list` argument:

- `va_start` — which initializes the list;
- `va_arg` — which returns the next argument in the list;
- `va_end` — which cleans up the variable argument list.

The following code illustrates the use of a variable argument list function. Just like the `main()` prototyped as `int main(int argc, char *argv[])`, where `argc` is the argument count, whereas `argv` is the argc-length list of argument values, here, too, even though the function definition does *not* specify the list of arguments, the function call *does*: the 1st argument is the number of arguments, and the following arguments are the argument values the function works with.

```
#include <stdarg.h> /* for va_start() and va_end() */
#include <stdio.h>

double average(int num, ...);

int main(void)
{
     printf("%f\n", average(3, 12.2, 22.3, 4.5));
     printf("%f\n", average(5, 3.3, 2.2, 1.1, 5.5, 3.3));
return 0;
}

/* this function will take the number of values to average
   followed by all of the numbers to average */
double average(int num, ...)
{
     va_list arguments;
     double sum=0;
     va_start(arguments, num);
     for (int x=0; x<num; x++)
     {
          sum+=va_arg(arguments, double);
     }
     va_end(arguments);
return sum/num;
}
```

The following program reads a variable number of strings (names) from the command line, and provides them as arguments to a function that reprints them in an (insufficiently) formatted table. (Yes, homework assignment is coming…)

```
#include <stdio.h>
#include <stdarg.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h> /* for malloc() */

#define MAXNAMES 10

char **MakeList(int num);
void MakeTable(int num, ...);

int main(void)
{
     char **list=MakeList(2);
     MakeTable(2, list[0], list[1]);
return 0;
}

char **MakeList(int num)
{
     char **names=malloc(MAXNAMES*BUFSIZ*sizeof(char));
     printf("Enter names, EOL separated:\n");
     for (int i=0; i<num; i++)
     {
          char *buf=malloc(BUFSIZ);
          names[i]=fgets(buf, BUFSIZ, stdin);
          int len=strlen(names[i]);
          if (names[i][len-1]=='\n')
               names[i][len-1]='\0';
     }
return names;
}
```

```
void MakeTable(int num, ...)
{
     va_list arguments;
     va_start(arguments, num);

     for (int i=0; i<num; i++)
     {
          printf("| %s |\n", va_arg(arguments, char *));
     }
}
```

Formatted input is provided by the family of `scanf()` functions:

```
#include <stdio.h>

int scanf(const char *format, …);

int fscanf(FILE *fp, const char *format, …);

int sscanf(const char *buf, const char *format, …);
```

All return number of input items assigned, EOF if error.

The following function returns a file descriptor from a `FILE *` object:

```
#include <stdio.h>

int fileno(FILE *fp);
```

Returns the file descriptor associated with the stream.

The source file above can be modified to use `scanf()` instead of `fgets()`:

```
#include <stdio.h>
#include <stdarg.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h> /* for malloc() */

#define MAXNAMES 10

char **MakeList(int num);
void MakeTable(int num, ...);
```

```
int main(void)
{
     char **list=MakeList(2);
     MakeTable(2, list[0], list[1]);
return 0;
}

char **MakeList(int num)
{
     char **names=malloc(MAXNAMES*BUFSIZ*sizeof(char));
     printf("Enter names, EOL separated:\n");
     for (int i=0; i<num; i++)
     {
          char *buf=malloc(BUFSIZ);
          //names[i]=fgets(buf, BUFSIZ, stdin);
          scanf("%s", buf);
          names[i]=buf;
          int len=strlen(names[i]);
          if (names[i][len-1]=='\n')
               names[i][len-1]='\0';
     }
return names;
}

void MakeTable(int num, ...)
{
     va_list arguments;
     va_start(arguments, num);

     for (int i=0; i<num; i++)
     {
          printf("| %s |\n", va_arg(arguments, char *));
     }
}
```

Temporary Files

The following functions allow the creation of a temporary file:

```
#include <stdio.h>

char *tmpnam(char *ptr);
```

Returns pointer to pathname.

```
#include <stdio.h>

FILE *tmpfile(void);
```
Returns file pointer if OK, `NULL` if error.

`tmpnam()` returns a uniquely generated filename, up to TMP_MAX (defined in `<stdio.h>`) characters. If *`ptr`* is NULL, the generated pathname is stored in a static area (meaning the pointer to the pathname is "remembered" across calls to `tmpnam()`, resulting in the pathname being released and reused for rewriting by another process). This definition is deprecated.

The following current implementations are used nowadays:

```
#include <stdio.h>

int mkstemp(char *template);
```
Returns a file descriptor if OK, -1 if error.

```
char *mktemp(char *template);
```
Returns a filename if OK, `NULL` if error.

The `mktemp()` function takes the given file name template and overwrites a portion of it to create a file name. This file name is guaranteed not to exist at the time of function invocation and is suitable for use by the application. The template may be any file name with some number of Xs appended to it, for example /tmp/temp.XXXXXX. The trailing Xs are replaced with a unique alphanumeric combination. The number of unique file names `mktemp()` can return depends on the number of Xs provided; six Xs will result in `mktemp()` selecting one of 56800235584 (62 ** 6) possible temporary file names.

The `mkstemp()` function makes the same replacement to the template and creates the template file, mode 0600, returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

The following program creates two temporary files via pathname and via file descriptor:

```
#include <stdio.h>
#include <stdlib.h> /* for mkstemp() */
#include <fcntl.h> /* for open() */
#include <sys/stat.h> /* for O_RDWR */

int main(void)
{
     //char temp_filename[L_tmpnam];
     char *temp_filename;

     char template[]="/Users/awise/Stevens/Lecture7/
temp.XXXXXX";
     temp_filename=mktemp(template);
     printf("temp_filename=%s\n", temp_filename);
     int fd=open(temp_filename, O_RDWR | O_CREAT);
     printf("Using mktemp(): fd=%d\n", fd);

     char template1[]="/Users/awise/Stevens/Lecture7/
temp.XXXXXX";
     int fd1=mkstemp(template1);
     printf("Using mkstemp(): fd=%d\n", fd1);
     FILE *file1=tmpfile();
return 0;
}
```

…And this is the example from APUE:

```
#include <stdio.h>

#define MAXLINE 1024

int main(void)
{
     char name[L_tmpnam], line[MAXLINE];
     FILE *fp;

     printf("%s\n", tmpnam(NULL));
     tmpnam(name); /* name is assigned here */
     printf("%s\n", name);
     if ((fp=tmpfile())==NULL)
          printf("tmpfile() error\n");

     fputs("one line of output\n", fp);
     rewind(fp);
     if (fgets(line, sizeof(line), fp)==NULL)
          printf("fgets() error\n");
     fputs(line, stdout);
exit(0);
}
```

Function Pointers

Functions can be passed as arguments to other functions. A function's name is a pointer, and can be assigned to a void pointer variable, taking an argument of the same type as the function we wish to pass as an argument. The reason for passing a function as an argument, is that sometimes in development it is necessary to write function whose behavior can change, but whose activity can be passed to other functions. The following simple example shows a function being passed to another function.

```
#include <stdio.h>

void func1(int n);
void dispatch(void (*funcptr)(int), int);

int main(int argc, char **argv)
{
     int a=10;
     void (*fp)(int); /* declare pointer to a function that
takes an int */
     fp=func1;

     dispatch(fp, a);
return 0;
}

void func1(int n)
{
     printf("func1() prints %d\n", n);
}

void dispatch(void (*funcptr)(int), int arg)
{
     printf("dispatch() calls func1(%d)\n", arg);
     (*funcptr)(arg);
}
```

Just as we can typedef a variable type, we can typedef a function:

```
#include <stdio.h>

void func1(int n);
void dispatch(void (*funcptr)(int), int);

int main(int argc, char **argv)
{
     int a=10;
     typedef void (*fp)(int); /* typedef a pointer to a function 
that takes an int */
     fp address=func1;

     dispatch(address, a);
return 0;
}

void func1(int n)
{
     printf("func1() prints %d\n", n);
}

void dispatch(void (*funcptr)(int), int arg)
{
     printf("dispatch() calls func1(%d)\n", arg);
     (*funcptr)(arg);
}
```

The output from both of these functions is:

```
$ ./myfunctionpointer
dispatch() calls func1(10)
func1() prints 10
```

Homework (due Tuesday, Mar-1-2016): Write a cron job that appends the current date and time into a temporary file. The output should be in a formatted table.