CSC 181F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC181F in the Fall term 1999/2000 at the University of Toronto

Reading Assignment

K.N. King   Sections 19.1 , 19.2, 19.3

Supplementary reading

S. McConnell   Chapter 6

Modules

- An **important** mechanism for packaging related declarations (constants, types and variables) and functions
- Modules are **the way** to build large software systems
- Modules can be used as *plug-replaceable* atomic units in larger software systems.
- C was designed before modules were recognized as a valuable programming tool. The **class** construct in C++ adds modules to C.
- **Learning to build large software systems using modules (or classes in C++) is an important programming skill**

Abstract Data Types

- One good way of thinking about modules is that they provide *abstract data types*
- A module provides:
  - a way of defining an abstract type
  - a set of operations on that type
    including creation, manipulation, destruction.
- Examples:
  - Complex Numbers
  - Set
  - Stack

Information Hiding and Encapsulation

- One **really important** aspect of modules is **information hiding.** A module hides (*encapsulates*) data that is declared inside the module from the rest of the program
- A module allows access to this hidden data via its exported functions which are the modules *interface* to the rest of the program
- Outside of a module, the rest of the program does not know the actual representation of the data items declared in the module (e.g., array, structure, union, list, tree etc.)
- Hiding the representation of the data gives the author of the module the freedom to change the internal workings of the module with an *absolute guarantee* that other parts of the program can't detect the changes (as long as the module's interface doesn't change

Simulating Modules in C

- C does not provide any construct for modules, i.e. a mechanism to achieve *information hiding*. By careful design and programmer discipline most of the desirable attributes of modules can be simulated in C by following the rules listed below
  - Always use a `.h` file and a `.c` file to represent the module
  - The `.h` should define the complete interface to the module *and nothing else*. The `.c` file should contain the implementation of the module.
  - All items in the `.h` file should have the **extern** attribute
  - Always **#include** the `.h` file in the `.c` file to force the complier to check the `.c` for consistency with the `.h` file.
  - Data and functions that are logically private (internal only) to the module should be declared with the **static** attribute.
  - In the interface file, declare types that you want to remain hidden as type **void** or **void *** . The real internal representation of these types is only used in the `.c` file that implements the interface.

HOW TO Handle Module Data

- Assuming a module implements some *abstract data type*.
- A *single instance module* has *one* instance of the data declared inside the module. The functions exported from this module all manipulate this one data structure.
  Example: module that provides a table or database.
- A *multiple instance module* exports a type that allows users of the module to create as many instances as they want of the abstract data type. The users instances of the abstract data type are passed to the functions exported by the module. **Examples:** Set, Complex.
- With good information hiding a client of a multiple instance module can create variables of the abstract data type but cannot examine or use the internal represention of the data.

Module Example – Complex

```
/* File complex.h */
/* Interface to Complex Module */

typedef void * Complex ;
extern double compRe(const Complex C ) ;
extern double compIm(const Complex C ) ;

extern Complex compAdd(const Complex C1 , const Complex C2) ;
extern Complex compSub(const Complex C1 , const Complex C2) ;
extern Complex compMul(const Complex C1 , const Complex C2) ;
extern Complex compDiv(const Complex C1 , const Complex C2) ;

extern Complex compCreate(double real , double imag) ;
extern void compPrint(const Complex C) ;
```

Module Example – Complex

```
/* File complex.c */
/* Complex Module implementation*/
#include "complex.h"
static struct complexType {
    double real ;
    double imag ;
} ;
typedef struct complexType * realComplex ;

double compRe(const Complex C)
{
    return ((realComplex) C)−> real ;
}

double compIm(const Complex C) {...}

Complex compAdd(const Complex C1 , const Complex C2)
{
    realComplex C = (realComplex) malloc(sizeof(struct complexType)) ;

    assert(C != NULL) ;
    C−> real = ((realComplex) C1)−> real + ((realComplex) C2)−> real ;
    C−> imag = ((realComplex) C1)−> imag + ((realComplex) C2)−> imag ;

    return (Complex) C ;
}
...
```


Module Example – Stack

- Stack is a linear list that can be accessed at just one of its ends
  - Consider a stack of plates:
    You add a plate on top of the stack
    You remove a plate from on top of the stack
- LIFO (Last In First Out)
- Operations on Stack
  - create(S): to bring existence an empty stack S
  - push(S, R): to add the object R to the top of the stack S
  - pop(S): to remove the object at the top of stack S
  - top(S, T): to assign to T the value of the object at the top of stack S without removing the object from the stack
  - empty(S): to ascetain if the stack S is empty or not

Module Example
Stack of integers

```
/* File stack.h */
/* Interface to Stack Module */

typedef void * Stack ;
typedef int StackElem ;

extern void push( Stack S , StackElem v) ;
extern void pop( Stack S ) ;

extern int isEmpty( Stack S );
extern StackElem top( Stack S );

extern Stack create() ;
```

```
/* File stack.c */
/* Stack Module implementation */
#include "stack.h"
static struct stackType {
    int sPtr ;
    StackElem sData[100] ;
} ;
typedef stackType * realStack ;
static stackCheck( realStack R ) { .. } ;
void push( Stack S , StackElem v ) { .. } ;
void pop( Stack S ) { .. } ;
int isEmpty( Stack S ) { .. } ;
StackElem top( Stack S ) { .. } ;
Stack create() { .. } ;
```


Example – Using the Stack Module

```
#include "stack.h"
...
  Stack stack1, stack2;
  StackElem temp;

  stack1 = create();
  /* Read integers from input and stack them */
  while (scanf("%d",&temp) != EOF)
    push(stack1, temp);

  stack2 = create();
  /* Print input in reverse order, copy to stack2 */
  while (!isEmpty(stack1)) {
    temp = top(stack1);
    pop(stack1);
    printf("%d\n", temp);
    push(stack2, temp);
  }
...
```

Stack of integers
Alternative Implementation

```
/* File stack.h */
/* Interface to Stack Module */

typedef void * Stack ;
typedef int StackElem ;

extern void push( Stack S , StackElem v) ;
extern void pop( Stack S ) ;

extern int isEmpty( Stack S );
extern StackElem top( Stack S );

extern Stack create() ;
```

```
/* File stack.c */
/* Stack Module implementation */
#include "stack.h"
static struct stackType {
    StackElem data ;
    struct stackType * next ;
} ;
typedef stackType * realStack ;
static stackCheck( realStack R ) { .. } ;
void push( Stack S , StackElem v ) { .. } ;
void pop( Stack S ) { .. } ;
int isEmpty( Stack S ) { .. } ;
StackElem top( Stack S ) { .. } ;
Stack create() { .. } ;
```


Reading Assignment

```
    K.N. King          Chapter 13

    K.N. King          Sections 23.4, 23.5

Supplementary reading

    Harbison & Steele  Chapter 12,  13, 14
```


Strings

char identifier [*size*] **;**

char * identifier **;**

- Strings are ultimately arrays of characters
  All good strings are null terminated by a character containing the value zero. ('\0')
  All string processing depends on this property
- **The programmer must allocate enough space for each string variable including space for terminating null**. Compiler allocates space for string literals.
  A string literal is a pointer to the first character of the string.
  Most string processing is done using pointers to characters
- **String literals enclosed in double quotes (")**
  `"This is a sample string."`
  Characters enclosed in single quotes (')
  `'a'    'A'    '\012'`

String Declaration Examples

```
char  ch;    /* single character  */

char  A  = 'A';  /* initialized single char */

char  ca[10];  /* 10-character array */

char date[10] = {'O', 'c', 't', 'o', 'b', 'e', 'r', '\0'};
char oct[10] = "October" ;
   /* the remaining elements are given the value '\0' */

char  * sp ;     /* pointer to string */

const char *cmsg =  "Put Your Message Here" ;
       /* pointer to string constant */

char msg[] = "Contact Wortman Advertising for Rates";
         /* initialized char array */
```

HOW TO Use null Terminated Strings

- **All** string processing in C assumes that strings are properly null terminated. CHAOS will ensue if this convention is ever violated.
- The null termination is the only way in C to find the end of a string.
- The compiler automatically adds a terminating null to all string constants. e.g. Internally "ABC" is "ABC\0"
- All library string functions assume arguments their arguments are null terminated and produce a null terminated result.
  WARNING: make sure all arguments you pass to builtin string functions are properly null terminated

```
#define NULLCHAR ( ( char )  0 )
char bigString[ 1000 ] ;
bigString[0] = NULLCHAR ;
```


- **WARNING:** Losing the null termination on the end of a string will cause your program to CRASH.
- **WARNING: all** storage for strings MUST include space for the null termination character.
- Any processing of strings as array of characters MUST reusult in a properly null terminated string

```
/* Insurance Trick */
bigString[ 999 ] = NULLCHAR ;
```

- Strings look like this:

char example[11] = "TEST CASE" ;

T	E	S	T		C	A	S	E	\0	
0	1	2	3	4	5	6	7	8	9	10

Operations on Strings

- *Length* - Get current length of string.
- *Copy* - Assign a new value to a string variable.
- *Append* - Add information to the end of a string.
- *Substring* - Select a sequence of characters from a larger string.
- *Concatenate* - Add one string to the end of another.
- *Character Search* - Search through a string looking for a given character.
- *String Search* - Search through a string looking for some other string.

String builtin functions
#include <string.h>

strcpy	strncpy	copy string
strcat	strncat	concatenate string
strcmp	strncmp	compare strings
strlen		length of string
strchr		search for character
strstr		search for substring
memmove		safe string copy

Use string builtin functions wherever possible

There are many more string functions see King Appendix D or Harbison & Steele for a complete list.

String Length Function

size_t **strlen**(**const char** * S) **;**

- size_t is an unsigned integer type defined in stddef.h.
- **strlen** returns the number of characters in S up to, but not including, the first null character.
- **WARNING: strlen** is a *SLOW* function in C. It must search the string S to the terminating null character to determine its length

Example:

```
int len;
char str[100];
len = strlen("abc");     /* len is now 3 */
len = strlen("");     /* len is now 0 */
len = sizeof(str);     /* len is now 100 */
```


String Copying Function

char * strcpy(**char** * dest **, const char** * src) **;**

char * strncpy(**char** * dest **, const char** * src **,** size_t N) **;**

- **strcpy** copies the string src into the string dest.
  if strlen(src) >= **sizeof(dest) an ERROR occurs and some random piece of memory gets trashed.**
- **strncpy** copies exactly N characters to dest.
  - If strlen(src) is less than N dest is filled out with null characters.
  - If strlen(src) is greater than N, only N characters are copied to dest **and dest is NOT null terminated.**
  - dest is null terminated if and only if strlen(src) $<$ N

- **Good Technique:** If there is **any** doubt about whether strlen (src) could be greater than sizeof(dest) use strncpy instead of strcpy.
- **WARNING:** strcpy and strncpy are NOT guaranteed to work correctly if src and dest overlap in memory. Use memmove instead.
- NOTE the assignment operator cannot be used to copy strings. You *must* use strcpy or strncpy.

```
char * S1 , * S2 ;
char S3[50] ;
S1 = "Test String" ;     /* Pointer Assignment */
S2 = S3 ;     /* S2 now points at S3 */
S3 = "BAD EVIL" ;     /* ERROR S3 is an array pointer constant */
strcpy( S3 , "MUCH BETTER" ) ;
```


Memory Copy Functions

void * memmove(**void** * dest, **const void** * src **,** size_t len) **;**

void * memcpy(**void** * dest, **const void** * src **,** size_t len) **;**

- **memmove** Copies *len* characters from src to dest.
  memmove will work correctly even if src and dest overlap in memory.
- **memcpy** copies *len* characters from src to dest.
  memcpy is not guaranteed to work correctly if src and dest overlap in memory but it is generally faster than memmove.
  WARNING: Use of memcpy is dangerous, use memmove instead.
- **WARNING:** an ERROR will occur and memory WILL get trashed if dest is not large enough to hold the result.

String Concatenation Function

char * **strcat**(**char** * dest, **const char** * src) **;**

char * **strncat**(**char** * dest, **const char** * src **,** size_t N) **;**

- **strcat** appends the contents of the string src to the *end of* the string dest.
- **strncat** copies *up to* N characters from src to the *end of* dest.
  If strlen(src) = N then N + 1 characters will be written to dest.
- **WARNING:** an ERROR will occur and memory will get trashed if dest is not large enough to hold the result of the concatenation.
  sizeof (dest) $>$ strlen(dest) + strlen(src) + 1

String Concatenation Examples

```
char str1[10], str2[10] ;

strcpy(str1, "abc") ;
strcat(str1, "DEF") ;
/* str1 now contains "abcDEF" */

strcpy(str1, "abc") ;
strcpy(str2, "DEF") ;
strcat(str1, str2 + 1) ;
/* str1 now contains "abcEF" */
```


String Comparison Function

int strcmp(**const char** * s1 **, const char** * s2) **;**

int strncmp (**const char** * s1 **, const char** * s2 **,** size_t N) **;**

- **strcmp** compares the strings s1 and s2, returns 0 if they are equal, negative number if s1 is less than s2, and positive number if s1 is greater than s2.
- **strcmp** compares strings using lexicographic ordering depending on the character encoding scheme.
- Comparison proceeds from the 1st character to the last until a character mismatch occurs or the end of a string is encountered.
  If a mismatch is found, result is determined by character comparison using the order of characters.
  If no mismatch is found then
  - if the string values are of the same length, they are equal
  - if the string values are not of the same length, the longer string value is greater

String Comparison Examples (ASCII)

```
strcmp("a","A") > 0
strcmp("ABC","DEFG") < 0
strcmp("DEF","ABC") > 0
strcmp("aaaa","aaaA") > 0
strcmp("0","a") < 0
strcmp("CsC180F","CsC181F") < 0
strcmp("WXYZ","WXZZ") < 0
strcmp("pqrst","pqrs") > 0
strcmp("jklmnop","jklmnop") = 0
```

Note that strings **cannot be compared directly using relational operators** such as ”abc” $>$ ”abcd”.

Recall that string literal is a pointer to the first character of the string.

Character Search Function

char * **strchr**(**const char** * S **, int** C) **;**

char * **strrchr** (**const char** * S **, int** C) **;**

- **strchr** searches the string S for first occurrence of the character C.
- If C is found in S, a pointer to this first occurrence is returned, otherwise a NULL pointer is returned.
- **strrchr** performs the same comparison except that it returns a pointer to the *last* occurrence of C in S.
- The null terminating character *is* considered to be a part of the string, so
  T = strchr(S **,** NULLCHAR) **;**
  returns a pointer to the end of S

Substring Search Function

char * **strstr**(**const char** * src **, const char** * sub) **;**

- **strstr** searches the string src for first occurrence of the string sub.
- If sub is found in src, a pointer to this first occurrence is returned, otherwise a NULL pointer is returned.

Reading and Writing Strings

- Writing strings: **printf** writes the characters in a string one by one until it encounters a null character

```
char str[120] = "Are we having fun yet?" ;
printf ("Value of str:  %s\n", str) ;
```

- Reading strings: **scanf** skips white space and then read characters and stores them until it encounters a white space. **scanf** adds null character at the end of the string

```
scanf ( "%s", str ) ;
```

- **WARNING:** the character array given as an argument to **scanf MUST** be large enough to hold **any** possible input value. Otherwise you have an error in your program

- Reading full line: **gets** does not skip white space. **gets** reads until it finds a new line character, discards it, and add the null character at the end.

```
gets ( str ) ;
```

- **WARNING:** the character array given as an argument to **gets MUST** be large enough to hold **any** possible input value. Otherwise you have an error in your program **and some evil person can trash your program**
- **Good Technique:** If you can't always guarantee valid input, at least detect when your program has been hosed.

```
#define BUFFER_SIZE 256
char buffer[ BUFFER_SIZE + 1 ] ;
...
gets ( buffer ) ;
assert ( strlen( buffer ) <= BUFFER_SIZE ) ;
```

HOW TO Process Strings Efficiently

- Try to process strings wholesale rather than retail
- Try to avoid slow operations like **strlen**.
- Try to minimize the number of times strings get copied or concatenated.
- Use pointers to access strings efficiently.
- Avoid special cases, try to find general algorithms.

```
/* Exmaple of Gross Inefficiency */
char S[256] , T[256] ;
int J ;
/* Assume S given a value here */
/* copy S to T */
for ( J = 0 ; J <= strlen(S) ; J+ + )
    T[ J ] = S[ J ] ;
```


String Processing Templates

```
char * S , * T ;
/* Assume s initialized */
for ( T = S ; * T ; T+ + ) {
    /* process * T */
} ;

T = S ;
while ( * T ){
    /* process S */
    /* increment T */
} ;
```

```
for ( T = S + strlen(S) -1 ; T >= S ; T− − ){
    /* process * T */
} ;

T = S ;
do {
    /* process S */
    /* increment T */
while ( * T ) ;
```


Example - remove leading blanks

```
char * S , * T ;
/* Assume S initialized here */
/* using while loop */
while ( * S && * S == ' ' )
    strcpy( s, s+1 ) ;     /* Shift over one blank */
                           /* Grossly INEFFICIENT */
```

```
/* using for loop */
for ( T = S ; * T && T == ' ' ; T+ + )
    ;     /* Find 1st non blank */
strcpy( S , T ) ;     /* Shift over all blanks */
```


Example - remove trailing blanks

```
char * S , * T ;
/* Assume S initialized here */
/* Using while loop */
T = S + strlen(S) - 1 ;
while ( T > S && * T == ' ' )
    T− − ;
*(T+1) = NULLCHAR ;
```

```
/* Using for loop */
for ( T = S + strlen(S) - 1 ; T > S && * T == ' ' ; T− − )
    ;     Find last non blank */
*(T+1) = NULLCHAR ;
```

Example - remove all blanks

```
char * S , * T ;
char * Send ;
/* Assume S initialized here */

Send = strchr( S, NULLCHAR ) ;
WHILE( T = strchr( S, ' ' )){
    memmove( T , T + 1 , Send - T ) ;
    Send− − ;
}
```


Pattern Match & Substitution
General Case

In string S, search for first occurrence of pattern P.

If P is found, replace it with string R.

Cases	Action	Example
P not in S	Do nothing	SSSSSSSS
strlen(R) < strlen(P)	Shift end of S left so R just fits	SSRRRS
strlen(R) = strlen(P)	Exact replace of P with R	SSRRRSSS
strlen(R) > strlen(P)	Shift end of S right to make room for R	SSRRRRRSSS
P is empty string	Add R at start of S	RRRSSSSSSSS
R is empty string	Delete P from S	SSSSS
S is empty string	Do nothing	SSSSSSSS

Pattern Match & Substitution
General Case - CSC180F Solution

```
void replace( char * S , const char * P , const char * R ) {
    /* Replace P in S with R */
    char * T , * out ;
    for ( T = S ; T < S + strlen(S) - strlen(P) ; T+ + )
        if ( ! strncmp( T, P, strlen(P) ) {   /* Found P in S starting at T */
            out = (char * ) malloc( strlen(S) - strlen(P) + strlen(R) +1 ) ;
            assert ( out ) ;
            * out = NULLCHAR ;        /* build output */
            strncat( out, S, T - S ) ;
            strcat( out, R ) ;
            strcat( out, T + strlen( P ) ) ;
            strcpy( S , out ) ;
            free( out ) ;
            return ;
        }
}
```


Pattern Match & Substitution
General Case - CSC181F Solution

```
void replace( char * S , const char * P , const char * R ) {
    char * Pstart ;
    int Sleng, Pleng, Rleng ;
    if (( Pstart = strstr( S, P )) != NULL) {
        Sleng = strlen( S ) ;
        Pleng = strlen( P ) ;
        Rleng = strlen( R ) ;
        memmove( Pstart + Rleng , Pstart + Pleng , Sleng - (( Pstart - S) + Pleng ) + 1 ) ;
        strncpy( Pstart, R, Rleng ) ;
    }
}
```

Reading Assignment

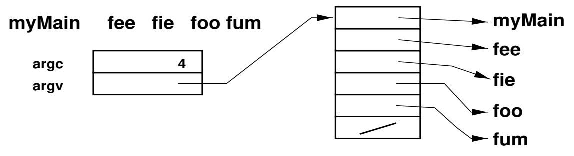
K.N. King   Chapter 3, 22

K.N. King   Sections 13.7

Main Program Revisited

void main(**int** *argc* **, char** * *argv*[])

- *argc* is the number of command line arguments that have been passed to the main program
- *argv* is a *pointer to an array* of **strings** which are the arguments to the program
  - argv[0] is the name that was used to invoke the program.
  - argv[1] to argv[argc] are the arguments that were passed to the program.
  - argv[argc + 1] is always NULL



Example of Argument Processing

```
main( int argc , char argv[] ) {
    /* Argument Processing  */
    for (argc--, argv++; argc > 0; argc--, argv++) {
            /* process options marked with -  */
        if (**argv == '-') {        /* A flag argument */
                /* Process all flags in this arg, e.g. -AbDf */
            while (*++(*argv)) {
                switch (**argv) {
                    case 'x':   /* process one flag */
                        ......
                        break;
                    default:
                        fprintf(stderr, "Unknown flag: '%c'; ", **argv);
                        exit(1);
                }
            }
        }
        else                /* Process everyting else */
            process( *argv );        /* Do something to argument */
    }
}
```


HOW TO Access Environment Information

#include $<$ stdlib.h $>$

char * getenv (**const char** * name) **;**

- In the Unix Shell you can set *environment variables* that can be used to communicate information to programs. Execute the Unix command

  `printenv`

  to see what your current environment variables are.
- The getenv function can be used by a program to retrieve the current value of environment variables from Unix.

  The argument *name* is the name of an an environment variable, getenv returns a pointer to a string containing the value courrent assigned to *name* and NULL if *name* is not defined.

  Example: Define TMPDIR=/bigSpace/tmp in environment.

Input and Output in C
#include < stdio.h >

- A *stream* is any source of input or any destination for output.
- A **file pointer** (of type FILE *) is the standard way to refer to streams in C programs.
- `stdio.h` defines three standard streams

File Pointer	Stream	Default Meaning
`stdin`	Standard Input	Keyboard input
`stdout`	Standard Output	Terminal Screen
`stderr`	Standard Error	Terminal Screen

  The command that invokes a program may **redirect** the meaning of the standard streams to be a file, a device or another program.

- Programs are usually written to work on the standard streams. If a program needs more sources or destinations it can create them as it's executing.

- The three standard streams are automatically ready to use when a program starts execution.
- By convention, standard error is used for printing error messages. These messages can be redirected to a destination that is different than standard output.

Text and Binary Files

- `stdio.h` supports two kinds of files.
- *Text Files* generally contain characters and are usually easily readable or printable. Text files are viewed as a sequence of *lines* where lines are separated by the NEWLINE character (`'\n'`).
- Binary files contain raw data in the encoded in the internal representation used by the hardware.
- Binary files can be used to store arrays, structures and other more complicated data structures.
- Generally binary input and output is much more efficient than text input and output since no conversion to/from the binary internal representation is required.

Character Input (Text Files)

int fgetc(FILE * stream)	Read one character from stream
int getc(FILE * stream)	Inline version of fgetc
int getchar(void)	getc(stdin)
char *fgets(**char** * S **,** **int** N **,** FILE * stream)	Reads at most N - 1 characters into array S
	Stops on newline. S is null terminated
char * gets(**char** * S)	Read next input line into S
	Like fgets(S **,** INFINITY **,** stdin)
int ungetc(**int** C, FILE * stream)	Push character c back onto stream

Character input functions return EOF on end of file or error.
WARNING: getc, fgetc and getchar return **int** not **char** .
String input functions return NULL on end of file or error.
WARNING: gets is inherently unsafe and should be avoided.
ungetc can be used to return *one* character to the stream.

Character Input Examples

```
int ch;
FILE * fp;
...
while ((ch = getc(fp)) != EOF) {
...
}

fgets(str, sizeof(str), fp); /* reads a line from fp */
fgets(str, sizeof(str), stdin);
   /* read a line from standard input */

#include <ctype.h>
...
while (isdigit(ch = getc(fp))) {
...
}
ungetc(ch, fp); /* puts back last value of ch */
```


HOW TO Handle Errors and End of File

- **Every call on an input function must check for possible end of file or error.**
- **WARNING:** The value of EOF returned by fgetc, getc and getchar is **NOT** a valid **char** value.
  Always read characters into an **int** variable, check for EOF and then assign the input to a **char** variable.
- The function
  int feof(FILE ***** stream)
  returns true if an end of file has been detected while reading stream. Note that this function does not return true *until* an attempt has been made to read **past** the end of file.

Character Output (Text Files)

int fputc(**int** C **,** FILE * stream)	writes character C on stream
int putc(**int** C **,** FILE STAR stream)	inline version of fputc
int putchar(**int** C)	putc(C **,** stdout)
int fputs(**const char** * S, FILE * stream)	writes string S on stream
int puts(**const char** * S)	fputs(S **,** stdout)

Output functions return EOF on error.

Examples:

```
FILE * fp ;
...
putchar( ch ) ; /* writes ch to stdout */
putc( ch, fp ) ; /* writes ch to fp */

puts( "Hello world!" ) ; /* writes to stdout */
fputs( "Hello world!" , fp ) ; /* writes to fp */
```


Formatted Output

- A *format string* is used to specify the exact layout of each line of printed output.
- The format string may contain text which is output literally
- The format string contains format control characters which cause data values to be printed.
- Embed ASCII control characters (e.g. `\n` (newline) , `\t` (tab) and `\f` (pagefeed)) in format string for line and page control
- **WARNING: Be Careful to exactly match format string with argument list. Mismatches cause crashes.**
- Sink for output can be standard stream, file or string

printf function

printf(**const char *** *format* , *expressionList*)

expressionList is a list of expressions separated by commas

format is a string that specifies formating for printed output

Expressions in the expression list are matched in order with format control characters in *format*

WARNING: most versions of C do NOT check that you've got this right. Be especially careful about char vs. string and different sizes of integers and reals.

Format Control Characters

%C

%−C

%*width*C

%.*prec*C

C is any format control character

A preceding − sign causes the field to be *left justified*

width and *prec* are integer constants. *width* specifies the printed width of the field in characters. *prec* specifies number of digits after decimal point for floating point numbers

width or precision can be * which causes value to be taken from next argument

Output Format Control Characters

c	single character
s	string
d	integer, signed decimal
i	integer, signed decimal
u	integer, unsigned decimal
o	integer, unsigned octal
x	integer, unsigned hexadecimal
X	integer, unsigned hexadecimal
p	void *, pointer
f	real, normal notation
e	real, scientific notation
E	real, scientific notation
g	real, shorter of e, f, notation
G	real, shorter of E, f, notation

Examples of formatted output

```
printf("CSC181F Assignment %d\n", aNumb );

sprintf( buffer, "(%s%c%s)",
           left_op, operator, right_op );

fprintf(myFile, "%-d %s %e\n",
          count, name, result );

printf("Pointer value is %p (%x)\n", p, p ) ;

printf("%.*f", 4, x);
printf("%.*f", WIDTH, x);
printf("%*.4f", 12, x);
```

fprintf & sprintf

fprintf(FILE ***** *stream*,**const char *** *format*, *expressionList*)
sprintf(**char *** S,**const char *** *format*, *expressionList*)

---

format and *expressionList* same as printf.
fprintf writes output to the designated file
sprintf writes output to the designated string

WARNING: Be Careful to always give a file as first argument to fprintf and a string as first argument to sprintf.
Make sure that string for sprintf is large enough to hold any possible result including a trailing null character.

HOW TO Use Files in C

- In a C program, a file is a *handle* that is used to access some external source or destination for data.
- The FILE ***** data type is a pointer to a control block that specifies the type of a stream, how to access the stream and the current location in the stream for input or output.
- The system establishes default streams for stdin, stdout and stderr. These can be overridden via redirection at the Unix command level.
- The fopen function is used to set up an association between an external file and a stream in the program.
- The fclose function is used to terminate the association set up by fopen.

File Open

FILE ***** fopen(**const char *** filename, **const char *** mode) ;

Opens named file and returns stream pointer or NULL if open fails.

Files must be opened before they are used for reading or writing.

Modes include:

”r”	open for reading
”w”	create text file for writing, discard previous contents
”a”	open for append or create
”r+”	open for update (read & write)
”w+”	create text file for update
”a+”	open or create for append & update

WARNING: Always check that fopen has returned a non-NULL pointer.

File Close

int fclose(FILE ***** stream)

Flush any unwritten data to output stream

Discard any unread input in input stream

Deallocates buffer, closes stream.

File should not be used after it is closed.

Returns zero if successful; otherwise returns EOF

scanf - formatted input

- User supplies format string, as in printf.
  Format string specifies order and format of input items.
- User provides *addresses of* variables in which to store input items.
- scanf attempts to read from its input source and match the input stream against the format string. Successful matches cause values to be assigned to the variables.
- scanf returns number of variables assigned to, *which may be less than the number of items requested*.

scanf function

int scanf(**char** * format, *varAddressList*)

---

format is a format string specifying the expected input format

varAddressList is a list of *addresses of* variables.
Use either pointers to allocated memory or & variable to generate the address of local variables.

Always check number of assignments done by scanf

Always provide a variable ADDRESS for each data item in the format list

scanf format string

- One more consecutive white-space in a scanf format string match zero or more white-space in the input stream.
- Ordinary characters are expected to match the characters in the input stream
- Conversion items beginning with % cause data values to be read and assigned to the next unassigned variable.
  An *assignment suppression character* * after the % causes a data item to be read by suppresses assignment to the variable. Use this to skip over unwanted input.
- scanf automatically skips over white-space (blank, tab, newline, carriage return, etc.) in its input

scanf conversion characters

Char	Type	Variable
d	decimal integer	**int** *
i	integer (any)	**int** *
o	octal integer	**int** *
u	unsigned integer	**unsigned** *
x	hex integer	**int** *
c	character (no ws skip)	**char** *
s	string	**char** *
e	float	**float** *
f	float	**float** *
g	float	**float** *

Conversion Modifiers

- Put `h` in front of `d, i, o, u` to indicate corresponding address is **short** *
- Put `l` in front of `d, i, o, u` to indicate corresponding address is **long** *
- Put `l` in front of `e, f, g` to indicate that corresponding address is **double** *
- `%` can be followed by an integer constant *width specifiers* that controls the number of input characters read. e.g `%1s` to read 1 character at a time

scanf examples

```
int i, j ;
long k ;
double X ;
char ch, str[100];
int nScanned ;

nScanned = scanf("%d%i" , & i , & j );

nScanned = scanf("%s is %c" , str ,  & ch );

nScanned = scanf("%*d%d" , & i);

nScanned = scanf("%le%ld", & X , & k );
```


fscanf and sscanf

int fscanf(FILE * stream, **char** * format , varAddressList)

int sscanf(**char** * source, **char** * format, varAddressList)

---

fscanf like scanf except input comes from designated file

sscanf like scanf except input comes from character string

Example:

fscanf(inFile , "%s" , str); /* read string from file */

sscanf(str , "%d%d" , &i , &j); /* extract two int from str */

HOW TO Use sscanf and ssprintf

- sprintf can be used in a program to build up compilcated character strings.
  - More efficient than using several strcpy/strcat operations.
  - Provides access to all of the builtin conversion routines from internal represeantation to characters.
- scanf can be used in a program to scan strings and extract information.
  - Provides access to the internal conversion routines.
  - Can read input with fgets and then try alternative analysis with different scanf calls. Allows you to validate input without crashing the program.
- **WARNING:** make sure string argument to sprintf is long enough to hold any possible output.

Block Input and Output

size_t fread(**void** * *ptr* **,** size_t *size* **,** size_t *nmemb* **,** FILE * *stream*)
size_t fwrite(**const void** * *ptr* **,** size_t *size* **,** size_t *nmemb* **,** FILE * *stream*)

---

fread reads an array from a stream to memory (ptr) in a single operation

fwrite copies an array from memory (ptr) to a stream in a single operation

ptr is the address of the data to transfer

size is the size of each element of an array in bytes

nmemb is the number of array elements to write/read

return value of fread is the number of elements read

return value of fwrite is the number of elements written

Good Technique: check return value against *nmemb*

HOW TO Use fread & fwrite

- Use fread and fwrite to move blocks of *binary* information between memory and disk files.
- fread and fwrite are *much* more efficient than fprintf and fscanf for moving large amounts of data to/from disk.
- The function setvbuf can be used to disable buffering when reading and writing binary information. This may improve speed.
- You can set *size* to total number of bytes to transfer and *nmemb* to one. This may be faster.

- **WARNING:** reading and writing any data structure containing pointers (including **char** *) will **NOT** produce a correct result.
- Example Test case

  Write 100,000 random double numbers to a file.

  fwrite was about 150 **times** faster then fprintf.

  Read 100,000 random double numbers from a file.

  fread was about 40 **times** faster than fscanf.

  Binary file was 800,000 bytes, text file was 1,300,000 bytes.

Example of Binary Input/Output

```
struct dStruct {
     double  xCoord, yCoord, zCoord ;
     long  redColor, greenColor, blueColor ;
} ;
struct dStruct  data[ 10000 ] ;
FILE * dataFile ;
...
assert( ( dataFile = fopen( "myFile", "w" ) ) );
fwrite( & data[ 0 ] , sizeof( struct dStruct ) ,
                      10000 , dataFile );
fclose( dataFile );
...
assert( ( dataFile = fopen( "myFile", "r" ) ) );
fread( & data[ 0 ] , sizeof( struct dStruct ) * 10000 ,
                      1 , dataFile );
fclose( dataFile );
```

Seek & Tell

long ftell(FILE * *stream*)

int fseek(FILE * *stream* **, long** *offset* **, int** *origin*)

---

ftell returns the current position in the file *stream*

fseek sets the file position for stream.
Position is set *offset* bytes from *origin*

origin is one of:

- SEEK_SET - start of file
- SEEK_CUR - current position
- SEEK_END - end of file

Use SEEK_SET with value from ftell
Use fseek(file **,** 0L **,** SEEK_CUR) to reset internal file pointers between read and write operations on the same file.

Example - reverse file in place

```
long  forward, back ;
FILE  *  revFile ;
/*  assume file contains  struct Data */
int DataSize = sizeof( struct Data );
struct Data forwardBuff, backwardBuff ;

assert( (revFile = fopen( "someFile", "rb+" ) ) );
forward = 0 ;
/* Find start of last structure in file */
fseek( revFile , DataSize , SEEK_END );`
back = ftell( revFile );
while( forward < back ) {
  fseek( revFile , forward , SEEK_SET );
  fread( & forwardBuff , DataSize , 1 , revFile );
  fseek( revFile, back, SEEK_SET );
  fread( & backwardBuff, DataSize, 1, revFile );
  fseek( revFile, back, SEEK_SET );
  fwrite( & forwardBuff, DataSize, 1, revFile );
  fseek( revFile, forward , SEEK_SET );
  fwrite( & backwardBuff, DataSize, 1, revFile );
  forward += DataSize ;
  back -= DataSize ;
}
```


Other FILE operations

freopen	reopen file, resetting internal pointers
fflush	flush output file buffers
remove	delete named file
rename	rename file
tmpfile	create temporary file with unique name
setvbuf	control file buffering

See Harbison & Steele for details.