INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

The United Nations University

SMR/774 - 2

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

REFERENCE MATERIAL ABOUT "C" AND GDB

Alvise NOBILE
International Centre for Theoretical Physics
P.O. Box 586
34100 Trieste
Italy

Reference material about

C

and

gdb

Appendix C

Library Routines

ANSI C requires each compiler to provide a set of standard library routines (i.e., macros and functions). These routines are part of the C environment. Moreover, each C compiler must provide the prototype declaration of each library function and the definition of each library macro in one of the standard header files (as specified in the ANSI C standard [ANSI 1988a]). These header files also contain the declarations of types and constants that are needed for using the library routines.

The standard header files that must be provided by every ANSI C compiler are

`assert.h`	`locale.h`	`stddef.h`
`ctype.h`	`math.h`	`stdio.h`
`errno.h`	`setjmp.h`	`stdlib.h`
`float.h`	`signal.h`	`string.h`
`limits.h`	`stdarg.h`	`time.h`

Standard header files may be included in any order. Multiple inclusions of these header files will not cause problems. Note that multiple inclusions of header files often occurs as a result of including files that include other files.

For some library routines, a compiler may provide both macro and function versions. To ensure that the macro version of a library routine is used, the library routine should not be declared explicitly. Instead, the appropriate header file should be included. Not including the appropriate header file rules out the use of the macro version of the library routine because header files contain the macro definitions. If the function version of a library routine is to be used, then the corresponding macro definition, if any, should first be explicitly removed by using the `#undef` instruction.

As an example, suppose that both function and macro versions of the library routine `atoi` are provided by the C compiler. The macro definition, if any, will be in the header file `stdlib.h`. The following paradigm ensures that macro version is used [ANSI 1988a]:

```
#include <stdlib.h>
...
i = atoi(str);
```

the definition of the macro `atoi`, can be used:

```
#include <stdlib.h>
#undef atoi
...
i = atoi(str);
```

Removing the macro definition, forces the linker to look for a function named `atoi` in the standard library which is then linked together with the rest of the program.

The description of each library routine consists of its syntactic specification and the description of its behavior or semantics. The syntactic specification of a C routine contains information that will allow the program to be compiled without error. By convention, the syntactic specification of a library routine consists of two parts:

1. One or more `#include` statements that include the header files containing the definition of the macro or the prototype declaration of the function implementing the routine, and declarations necessary to use the macro or function.
2. The function prototype declaration (an equivalent declaration is given in case of a macro).

When using a library routine, the `#include` statements specified in the specification of the routine should be given in the file containing references to the function (before the references).

For complete and detailed descriptions of these routines, please see the *American National Standard for Information Systems—Programming Language C* [ANSI C 1988a].

1. Diagnostic Routines

1.1 Macro `assert`

Include: `#include <assert.h>`
Prototype: `void assert(int expression);`

Behavior: `assert` is used to check whether or not the condition specified by `expression` is true. If `expression` is false, `assert` prints, on the standard error stream, diagnostic information such as the unsatisfied condition, and the name of the source file containing the `assert` call and its line number. It then terminates the program by calling `abort`.

2. Character Handling Routines

2.1 Function `isalnum`

Include: `#include <ctype.h>`
Prototype: `int isalnum(int c);`

Behavior: `isalnum` returns a nonzero value if `c` is a letter or a digit; otherwise, it returns zero.

2.2 Function `isalpha`

Include: `#include <ctype.h>`
Prototype: `int isalpha(int c);`

Behavior: `isalpha` returns a nonzero value if `c` is a letter; otherwise, it returns zero.

2.3 Function `iscntrl`

Include: `#include <ctype.h>`
Prototype: `int iscntrl(int c);`

Behavior: `iscntrl` returns a nonzero value if `c` is a control character; otherwise, it returns zero.

2.4 Function `isdigit`

Include: `#include <ctype.h>`
Prototype: `int isdigit(int c);`

Behavior: `isdigit` returns a nonzero value if `c` is a digit; otherwise, it returns zero.

2.5 Function `isgraph`

Include: `#include <ctype.h>`
Prototype: `int isgraph(int c);`

Behavior: `isgraph` returns a nonzero value if `c` is any printing character except the space (blank) character; otherwise, it returns zero.

2.6 Function `islower`

Include: `#include <ctype.h>`
Prototype: `int islower(int c);`

Behavior: `islower` returns a nonzero value if `c` is a lower-case letter; otherwise, it returns zero.

2.7 Function isprint

Include: `#include <ctype.h>`
Prototype: `int isprint(int c);`

Behavior: `isprint` returns a nonzero value if `c` is any printing character (including a space); otherwise, it returns zero.

2.8 Function ispunct

Include: `#include <ctype.h>`
Prototype: `int ispunct(int c);`

Behavior: `ispunct` returns a nonzero value if `c` is a punctuation character (but not a space); otherwise, it returns zero.

2.9 Function isspace

Include: `#include <ctype.h>`
Prototype: `int isspace(int c);`

Behavior: `isspace` returns a nonzero value if `c` is a white-space character (space, form-feed, new-line, carriage-return, horizontal-tab or vertical-tab); otherwise, it returns zero.

2.10 Function isupper

Include: `#include <ctype.h>`
Prototype: `int isupper(int c);`

Behavior: `isupper` returns a nonzero value if `c` is an upper-case character; otherwise, it returns zero.

2.11 Function isxdigit

Include: `#include <ctype.h>`
Prototype: `int isxdigit(int c);`

Behavior: `isxdigit` returns a nonzero value if `c` is a hexadecimal digit; otherwise, it returns zero.

2.12 Function tolower

Include: `#include <ctype.h>`
Prototype: `int tolower(int c);`

Behavior: If `c` is an upper-case character, then `tolower` returns the lower-case version of character `c`; otherwise, it returns `c`.

2.13 Function toupper

Include: `#include <ctype.h>`
Prototype: `int toupper(int c);`

Behavior: If `c` is a lower-case character, then `toupper` returns the upper-case version of `c`; otherwise, it returns `c`.

3. Mathematical Routines

3.1 Function acos

Include: `#include <math.h>`
Prototype: `double acos(double x);`

Behavior: `acos` returns the arc cosine of `x`, which must be in the range [−1, +1].[1] The arc cosine computed will be in the range [0, π] radians.

3.2 Function asin

Include: `#include <math.h>`
Prototype: `double asin(double x);`

Behavior: `asin` returns the arc sine of `x`, which must be in the range [−1, +1]. The arc sine computed is in the range [$-\pi/2$, $+\pi/2$] radians.

3.3 Function atan

Include: `#include <math.h>`
Prototype: `double atan(double x);`

Behavior: `atan` returns the arc tangent of `x`, which must be in the range [−1, +1]. The arc tangent computed is in the range [$-\pi/2$, $+\pi/2$] radians.

3.4 Function atan2

Include: `#include <math.h>`
Prototype: `double atan2(double y, double x);`

Behavior: `atan2` returns the arc tangent of `y/x`. The quadrant of the return value is determined by the signs of `x` and `y`. The arc tangent computed is in the range [$-\pi$, $+\pi$] radians.

---

1. The notation [*a*, *b*] specifies the interval from *a* to *b*, including the end values *a* and *b*. Using an opening (instead of [indicates that the end value *a* is not included in the interval. Similarly, using a closing) instead of] indicates that the end value *b* is not included in the interval.

3.5 Function cos

Include: `#include <math.h>`
Prototype: `double cos(double x);`

Behavior: `cos` returns the cosine of `x`, which must be in radians.

3.6 Function sin

Include: `#include <math.h>`
Prototype: `double sin(double x);`

Behavior: `sin` returns the sine of `x`, which must be in radians.

3.7 Function tan

Include: `#include <math.h>`
Prototype: `double tan(double x);`

Behavior: `tan` returns the tangent of `x`, which must be in radians.

3.8 Function cosh

Include: `#include <math.h>`
Prototype: `double cosh(double x);`

Behavior: `cosh` returns the hyperbolic cosine of `x`.

3.9 Function sinh

Include: `#include <math.h>`
Prototype: `double sinh(double x);`

Behavior: `sinh` returns the hyperbolic sine of `x`.

3.10 Function tanh

Include: `#include <math.h>`
Prototype: `double tanh(double x);`

Behavior: `tanh` returns the hyperbolic tangent of `x`.

3.11 Function exp

Include: `#include <math.h>`
Prototype: `double exp(double x);`

Behavior: `exp` returns e^x.

3.12 Function frexp

Include: `#include <math.h>`
Prototype: `double frexp(double value, int *exp);`

Behavior: `frexp` splits a floating-point number into a normalized fraction and an integral power of 2. It stores the integer power in `*exp` and returns a value `x`, such that `x` is in the interval [1/2, 1) or equal to zero, and `value` is equal to `x` multiplied by 2 to the power `*exp`.

3.13 Function ldexp

Include: `#include <math.h>`
Prototype: `double ldexp(double x, int exp);`

Behavior: `ldexp` returns `x` multiplied by 2^{exp}.

3.14 Function log

Include: `#include <math.h>`
Prototype: `double log(double x);`

Behavior: `log` returns the natural logarithm of `x`.

3.15 Function log10

Include: `#include <math.h>`
Prototype: `double log10(double x);`

Behavior: `log` returns the base-10 logarithm of `x`.

3.16 Function modf

Include: `#include <math.h>`
Prototype: `double modf(double value, double *iptr);`

Behavior: `modf` splits the argument `value` into integral and fractional parts (each has the same sign as `value`). It stores the integral part in `*iptr` and returns the fractional part.

3.17 Function pow

Include: `#include <math.h>`
Prototype: `double pow(double x, double y);`

Behavior: `pow` returns x^y.

3.18 Function sqrt

Include: `#include <math.h>`
Prototype: `double sqrt(double x);`

Behavior: `sqrt` returns the nonnegative square root of `x`.

3.19 Function ceil

Include: `#include <math.h>`
Prototype: `double ceil(double x);`

Behavior: `ceil` returns the smallest integer greater than or equal to `x`.

3.20 Function fabs

Include: `#include <math.h>`
Prototype: `double fabs(double x);`

Behavior: `fabs` returns the absolute value of `x`.

3.21 Function floor

Include: `#include <math.h>`
Prototype: `double floor(double x);`

Behavior: `floor` returns the largest integer less than or equal to `x`.

3.22 Function fmod

Include: `#include <math.h>`
Prototype: `double fmod(double x, double y);`

Behavior: `fmod` returns the floating-point remainder of `x/y`. i.e., it returns $x-iy$ for some integer i such that, if `y` is not zero, then the value returned is less than `y` and it has the same sign as `x`.

4. Nonlocal Jump Routines

Besides other items, header file `setjmp.h` contains the declaration of type `jmp_buf` which is used for saving and restoring environments in conjunction with the `setjmp` and `longjmp` routines.

4.1 Macro setjmp

Include: `#include <setjmp.h>`
Prototype: `int setjmp(jmp_buf env);`

Behavior: `setjmp` saves its calling environment in argument `env` for later use by `longjmp`. When returning from a direct invocation, `setjmp` returns zero. If it returns as a result of calling `longjmp`, then `setjmp` returns a nonzero value.

4.2 Function longjmp

Include: `#include <setjmp.h>`
Prototype: `void longjmp(jmp_buf env, int val);`

Behavior: `longjmp` restores the environment saved in `jmp_buf` by the last invocation of `setjmp`. If the function containing the `setjmp` invocation has completed, then the behavior of `longjmp` is undefined. After executing a `longjmp` call, program execution continues as if the corresponding invocation of `setjmp` had just returned the value `val`. Note that `longjmp` cannot make `setjmp` return a zero. If `val` is equal to zero, `setjmp` will return one.

5. Signal Handling Routines

Besides other items, header file `signal.h` contains definitions of the macros `SIG_DFL`, `SIG_ERR` and `SIG_IGN` which are used for handling signals; these macros are discussed later. File `signal.h` also defines following constants identifying signals:

signal name	signal generated due to
SIGABRT	an abnormal termination, e.g., one caused by calling `abort`
SIGFPE	an erroneous arithmetic operation, e.g., zero divide or an overflow
SIGILL	an illegal instruction
SIGINT	an interrupt, e.g., from the keyboard
SIGSEGV	an invalid memory reference
SIGTERM	a termination request sent to the program

These signals are generated automatically; they can also be generated by calling function `raise`.

5.1 Function signal

Include: `#include <signal.h>`
Prototype:
```
void (*signal(int sig,
              void (*fun)(int)))(int);
```

Behavior: `signal` associates the function `fun` (signal handler) with the signal numbered `sig`. How the signal numbered `sig` is handled depends upon the value of `fun`:

value of fun	signal handling
SIG_DFL	default handling
SIG_IGN	signal is ignored
pointer to function *f*	function *f* (*signal handler*) is called.

At program startup, signals may be ignored or handled in the default manner; this treatment is implementation dependent.

If `signal` executes successfully, then it returns the value of the previous signal handler for `sig`. Otherwise, `SIG_ERR` is returned and a positive value

is stored in `errno`.

5.2 Function `raise`

Include: `#include <signal.h>`
Prototype: `int raise(int sig);`

Behavior: `raise` generates signal `sig`. If successful, `raise` returns zero; otherwise, it returns a nonzero value.

6. Macros to Handle Variable Number of Arguments

Besides other items, header file `stdarg.h` contains the definitions of type `va_list` and the macros `va_start`, `va_arg` and `va_end`, which are used for accessing arguments of a function that can be called with a variable number of arguments. Information needed by the variable argument manipulation macros is stored in an object of type `va_list`.

The variable number of arguments, which do not have explicit names, is indicated by the ellipsis following the last parameter in the function header. The rightmost parameter of such a function is special (the one just before the ellipsis) and is designated as $parm_n$ in the discussion below. Parameter $parm_n$ must not be given the `register` storage class, or be a function or an array type or a type incompatible with the argument type (after the default argument promotions).

6.1 Macro `va_start`

Include: `#include <stdarg.h>`
Prototype: `void va_start(va_list ap,` $parm_n$`);`

Behavior: `va_start` must be invoked before invoking `va_arg` to access the unnamed arguments (represented by the ellipsis). `va_arg` initializes `ap`.

6.2 Macro `va_arg`

Include: `#include <stdarg.h>`
Prototype: *type* `va_arg(va_list ap,` *type*`);`

Behavior: The i^{th} invocation of `va_arg` (after invoking `va_start`) returns the value of argument $parm_{n+i}$. Parameter `ap` must be the one initialized by `va_start`. Parameter *type* specifies the type of the next argument.

6.3 Macro `va_end`

Include: `#include <stdarg.h>`
Prototype: `void va_end(va_list ap);`

Behavior: `va_end` must be called after accessing the variable arguments.

7. Input/Output Routines

Besides other items, header file `stdio.h` contains the definitions of the following types and macros:

types	macros	macros
`size_t`	`NULL`	`SEEK_CUR`
`FILE`	`_IOFBF`	`_END`
`fpos_t`	`_IOLBF`	`SEEK_SET`
	`_IONBF`	`TMP_MAX`
	`BUFSIZ`	`tmpnam`
	`EOF`	`stderr`
	`FOPEN_MAX`	`stdin`
	`FILENAME_MAX`	`stdout`
	`L_tmpnam`	

7.1 Function `remove`

Include: `#include <stdio.h>`
Prototype: `int remove(const char *fname);`

Behavior: `remove` deletes the file with the name pointed to by `fname`. If successful, `remove` returns zero; otherwise, it returns a nonzero value.

7.2 Function `rename`

Include: `#include <stdio.h>`
Prototype: `int rename(const char *old,`
`const char *new);`

Behavior: `rename` changes the name of a file from that pointed to by `old` to that pointed to by `new`. If successful, `rename` returns zero; otherwise, it returns a nonzero value.

7.3 Function `tmpfile`

Include: `#include <stdio.h>`
Prototype: `FILE *tmpfile(void);`

Behavior: `tmpfile` creates a temporary file (opened for update). This file is automatically removed when it is closed, or upon program termination. If successful, `tmpfile` returns a pointer to the file created; otherwise, it returns the null pointer.

7.4 Function tmpnam

Include: `#include <stdio.h>`
Prototype: `char *tmpnam(char *s);`

Behavior: `tmpnam` generates a new unique file name (string) every time it is called. If `s` is the null pointer, then `tmpnam` stores the file name in an internal static object and returns a pointer to this object. Otherwise, `tmpnam` puts the file name in the array pointed to by `s` and returns `s`. Note that the array size must be greater than or equal to `L_tmpnam` (defined in `stdio.h`).

7.5 Function fclose

Include: `#include <stdio.h>`
Prototype: `int fclose(FILE *stream);`

Behavior: `fclose` flushes the stream pointed to by `stream` and closes the associated file. If successful, `fclose` returns zero; otherwise, it returns `EOF`.

7.6 Function fflush

Include: `#include <stdio.h>`
Prototype: `int fflush(FILE *stream);`

Behavior: `fflush` flushes `stream`, i.e., it starts writing unwritten buffered output to the file associated with `stream`. If `fflush` is called with a null pointer, then all output streams are flushed. If successful, `fflush` returns zero; otherwise, it returns `EOF`.

7.7 Function fopen

Include: `#include <stdio.h>`
Prototype: `FILE *fopen(const char *fname, const char *mode);`

Behavior: `fopen` opens the file with the name pointed to by `fname` and associates a stream with it. The values of argument `mode`, which specifies the file mode, are listed below:

mode	explanation
"r"	Open text file for reading.
"w"	Create text file for writing. Existing files are truncated to zero length.
"a"	Open or create text file for appending.
"rb"	Open binary file for reading.
"wb"	Create binary file for writing. Existing files are truncated to zero length.
"ab"	Open or create binary file for appending.
"r+"	Open text file for update (reading and writing).
"w+"	Create text file for update. Existing files are truncated to zero length.
"a+"	Open or create text file for update and appending.
"r+b"	Open binary file for update (reading and writing).
"w+b"	Create binary file for update. Existing files are truncated to zero length.
"a+b"	Open or create binary file for update and appending.

Both input and output may be performed on a stream if the associated file has been opened with the update mode. However, input may follow output only after an intervening call to `fflush`, or to one of the file positioning functions `fseek`, `fsetpos` or `rewind`. Output may follow input only after calling a file positioning function, except if the end-of-file has been encountered.

If successful, `fopen` returns a pointer to the stream; otherwise, it returns the null pointer.

7.8 Function freopen

Include: `#include <stdio.h>`
Prototype:
```
FILE *freopen(const char *fname,
              const char *mode, FILE *stream);
```

Behavior: `freopen` opens the file with the name pointed to by `fname` and associates with it the stream pointed to by `stream`. Before doing this, `freopen` closes the file, if any, associated with `stream`. Argument `mode` is used as in function `fopen`.

If successful, `freopen` returns the value of `stream`; otherwise, it returns the null pointer.

7.9 Function setbuf

Include: `#include <stdio.h>`
Prototype: `void setbuf(FILE *stream, char *buf);`

Behavior: Calling `setbuf` is equivalent to calling `setvbuf` with `mode` equal to `_IOFBF` (`_IONBF` if `buf` is the null pointer) and `size` equal to `BUFSIZ`. The only difference is that `setbuf`, unlike `setvbuf`, does not return a value.

7.10 Function setvbuf

Include: `#include <stdio.h>`
Prototype:
```
int setvbuf(FILE *stream, char *buf,
                          int mode, size_t size);
```

Behavior: `setvbuf` is used to specify buffering of the stream pointed to by `stream` as indicated by parameter `mode`:

mode	effect
_IOFBF	fully buffered input/output
_IOLBF	line buffered output
_IONBF	unbuffered input/output

If `buf` is not the null pointer, the array it points to may be used to buffer the input/output (instead of an internally allocated buffer). `size` specifies the size of this array. `setvbuf` is used after associating an open file with a stream, but before performing input or output.

If successful, `setvbuf` returns zero; otherwise, it returns a nonzero value.

7.11 Function fprintf

Include: `#include <stdio.h>`
Prototype:
```
int fprintf(FILE *stream,
                    const char *fmt, ...);
```

Behavior: `fprintf` writes output to the stream pointed to by `stream` as specified by the string pointed to by `fmt` which contains characters to be printed and conversion specifications (formats) specifying how the arguments (indicated by the ellipsis) are to be printed.

Each format begins with the `%` character which is followed by

1. Zero or more *flags* modifying the format.
2. An optional decimal integer specifying the minimum *field width*. If necessary, the value printed is left padded (right padded if the left adjustment flag is given) with blanks.

3. An optional *precision* (a period followed by a decimal integer) which specifies the
   a. minimum number of digits to be printed for `d`, `i`, `o`, `u`, `x` and `X` formats,
   b. number of digits to be printed after the decimal point for `e`, `E` and `f` formats,
   c. maximum number of significant digits for `g` and `G` formats, or
   d. maximum number of characters to be printed for the `s` format.
4. An optional `h` (`l`) specifying that the following `d`, `i`, `o`, `u`, `x` or `X` format applies to a `short` (`long`) `int` or `unsigned short` (`long`) `int` argument, or that the following `n` format applies to a `short` (`long`) `int *` argument; or an optional `L` specifying that the following `e`, `E`, `f`, `g` or `G` format applies to a `long double` argument.
5. A character specifying the format.

An asterisk `*` given for the field width (precision) indicates that an argument specifying the field width (precision) will be given before the argument to be printed.

The flags that can modify the format are listed below:

flag	meaning
-	Left-justify when output.
+	Print a leading plus or minus sign for signed values.
space	If the first character of a signed value is not a sign, then print a leading space.
#	o format: increase the precision to force the first digit of the output to be zero. x (X) format: print a leading 0x (0X) for nonzero results. e, E, f, g and G formats: print a decimal point. g and G formats: do not remove trailing zeros.
0	d, i, o, u, x, X, e, E, f, g and G formats: use leading zeros for padding.

The formats and their meanings are

format	meaning
d, i, o, u, x, X	Print an `int` argument as a signed decimal (`d` or `i`), an unsigned octal (`o`), an unsigned decimal (`u`) or as an unsigned hexadecimal (`x` or `X`).
f	Print the `double` argument (rounded appropriately) in the style [-]*ddd.ddd*. The number of digits printed after the decimal point is specified by the precision (default is 6).
e, E	Print the `double` argument (rounded appropriately) in the style [-]*d.e±dd*. One digit is printed before the decimal point and the number of digits printed after it is specified by the precision (default is 6). An `e` (`E`) is printed before the (at least two-digit) exponent.
g, G	The `double` argument is printed in the style specified by the `f` or `e` (`E`) formats. The `e` (`E`) format is used only for exponents less than −4 or greater than or equal to the precision.
c	Print the `int` argument as an `unsigned char`.
s	Print the string pointed to by the argument. Characters up to (but not including) the terminating null character are printed. If *n* is the precision, then only *n* characters will be printed.
p	Print the pointer argument, which must be of type `void *`.
n	The corresponding argument must point to an integer into which is written the number of characters printed up to now by this `fprintf` call. Nothing is printed.
%	Print the % character.

Notation [*a*] is used to specify the optional occurrence of item *a*.

If successful, `fprintf` returns the number of characters printed; otherwise, it returns a negative value.

7.12 Function `fscanf`

Include: `#include <stdio.h>`

Prototype:
```
int fscanf(FILE *stream,
           const char *fmt, ...);
```

Behavior: `fscanf` reads input from the stream pointed to by `stream`, according to the formats specified in the string pointed to by `fmt`, and assigns the values read to the objects pointed to by the remaining arguments (indicated by the ellipsis), which must all be pointers.

Each format begins with the % character which is followed by

1. An optional * indicating that the input is to be read but not assigned to any object (no corresponding argument is given). This is equivalent to skipping over a data item.
2. An optional decimal integer specifying the maximum field width.
3. Formats `d`, `i`, `n`, `o` and `x` may be preceded by `h` (`l`) which indicates that the corresponding argument is a pointer to a `short` (`long`) `int` and not a pointer to `int`. Similarly, format `u` may be preceded by `h` (`l`) which indicates that the corresponding argument points to an `unsigned short` (`long`) `int` and not to an `unsigned int`. Finally, formats `e`, `f` and `g` may be preceded by a `l` (`L`) to indicate that the corresponding argument points to a (`long`) `double` and not a `float`.
4. A character specifying the format.

A white space in the format string indicates that input is to be read (but not assigned to any object) up to the first nonwhite-space character (which remains unread) or until no further characters can be read. An ordinary character in the format string indicates that the next input character is to be read only if it matches the specified character.

White-space characters are skipped in the input unless the next format specifier is a `[`, `c` or `n`. Input is read for each format except for the `n` format.

Unless an asterisk is given in the format to suppress assignment, the item read will be stored in the object pointed to by the corresponding argument.

The formats and their meanings are

format	meaning
d	Read a decimal integer; the corresponding argument must be an integer pointer.
i	Read an integer; the corresponding argument must be an integer pointer.
o	Read an optionally signed octal integer; the corresponding argument must be an integer pointer.
u	Read an unsigned decimal integer; the corresponding argument must be an unsigned integer pointer.
x	Read an optionally signed hexadecimal integer; the corresponding argument must be an integer pointer.
e, f, g	Read a floating-point integer; the corresponding argument must be a floating-point pointer.
s	Read a string, i.e., a sequence of nonwhite-space characters; the corresponding argument must be a pointer an array large enough to hold the string plus the automatically added terminating null character.
[	Read a string consisting of characters specified after the [and up to the terminating] (called the "matching set"); the corresponding argument must be a pointer an array large enough to hold the string plus the automatically added terminating null character. If the first character after the left bracket is a circumflex (^), then the characters read are those not in the matching set. As a special case, if the format begins with [ ] or [^], the right bracket is considered to be part of the matching set; the next] ends the matching set.
c	Reads *n* characters where *n* is the field width (default value is 1); the corresponding argument must be a pointer an array large enough to hold the string; a terminating null character is not added.
p	Read a pointer value (such as one printed with `fprintf`); the corresponding argument must be a `void` pointer.
n	No input is read; the corresponding argument must be a pointer to integer into which is written the number of characters read up to now by this `fscanf` call.
%	Matches a single %.

`fscanf` returns `EOF` if it fails before reading any data; otherwise, it returns the number of input items assigned (may not be the same as the number of items read).

7.13 Function `printf`

Include: `#include <stdio.h>`
Prototype: `int printf(const char *fmt, ...);`

Behavior: `printf` is similar to `fprintf` except that it writes to `stdout`.

7.14 Function `scanf`

Include: `#include <stdio.h>`
Prototype: `int scanf(const char *fmt, ...);`

Behavior: `scanf` is similar to `fscanf` except that it reads from `stdin`.

7.15 Function `sprintf`

Include: `#include <stdio.h>`
Prototype: `int sprintf(char *s, const char *fmt, ...);`

Behavior: `sprintf` similar to `fprintf`, except that it writes to an array (the first argument). A null character is written at the end of output; it is not included in the value returned by `sprintf`.

7.16 Function `sscanf`

Include: `#include <stdio.h>`
Prototype:
```
int sscanf(const char *s, const char *fmt,
                                    ...);
```

Behavior: `sscanf` is similar to `fscanf`, except that it reads from a string (the first argument). Reaching the end of the string is equivalent to encountering the end-of-file.

7.17 Function `vfprintf`

Include:
```
#include <stdarg.h>
#include <stdio.h>
```
Prototype:
```
int vfprintf(FILE *stream,
             const char *fmt, va_list arg);
```

Behavior: `vfprintf` is similar to `fprintf` except that the variable argument list is replaced by the argument `arg`, which contains information about variable arguments; `arg` must have been initialized by invoking `va_start` and it may have been used in subsequent `va_arg` invocations. Functions called with a variable number of arguments can pass a variable of type `va_list` holding information about the variable arguments to `vfprintf` for printing the variable arguments.

7.18 Function vprintf

Include: `#include <stdarg.h>`
`#include <stdio.h>`
Prototype: `int vprintf(const char *fmt, va_list arg);`

Behavior: `vprintf` is similar to `vfprintf` except that it writes to `stdout`.

7.19 Function vsprintf

Include: `#include <stdarg.h>`
`#include <stdio.h>`
Prototype: `int vsprintf(char *s, const char *fmt, va_list arg);`

Behavior: `vsprintf` is similar to `vfprintf`, except that it writes its output to a character array (specified by parameter `s`).

7.20 Function fgetc

Include: `#include <stdio.h>`
Prototype: `int fgetc(FILE *stream);`

Behavior: `fgetc` returns the next character from the input stream pointed to by `stream`. On encountering the end-of-file, `fgetc` sets the end-of-file indicator associated with `stream` and returns `EOF`. If a read error occurs, `fgetc` sets the error indicator associated with `stream` and returns `EOF`.

7.21 Function fgets

Include: `#include <stdio.h>`
Prototype: `char *fgets(char *s, int n, FILE *stream);`

Behavior: `fgets` reads up to `n-1` characters from the stream pointed to by `stream` into the array pointed to by `s`. After encountering a new-line character (which is stored in `s`) or the end-of-file, no further characters are read. A null character is written after the last character stored in `s`.

If successful, `fgets` returns `s`. Otherwise, if an end-of-file is encountered and no characters have been stored in `s`, or if a read error occurs, then `fgets` returns the null pointer.

7.22 Function fputc

Include: `#include <stdio.h>`
Prototype: `int fputc(int c, FILE *stream);`

Behavior: `fputc` writes character `c` to the output stream pointed to by `stream` and returns `c`. If a write error occurs, `fputc` sets the error indicator for `stream` and returns `EOF`.

7.23 Function fputs

Include: `#include <stdio.h>`
Prototype: `int fputs(const char *s, FILE *stream);`

Behavior: `fputs` writes the string pointed to by `s` (sans the terminating null character) to the stream pointed to by `stream`. If successful, `fputs` returns a nonnegative value; otherwise, it returns `EOF`.

7.24 Routine getc

Include: `#include <stdio.h>`
Prototype: `int getc(FILE *stream);`

Behavior: `getc` is similar to `fgetc` except that it may be implemented as a macro.

7.25 Function getchar

Include: `#include <stdio.h>`
Prototype: `int getchar(void);`

Behavior: `getchar` is similar to `getc` except that it reads from `stdin`.

7.26 Function gets

Include: `#include <stdio.h>`
Prototype: `char *gets(char *s);`

Behavior: `gets` reads characters from the input stream pointed to by `stdin` and stores them into the array pointed to by `s`. It reads characters until an end-of-file encountered or a new-line character is read. The new-line character is discarded; a null character is written after the last character stored in the array.

If successful, `gets` returns `s`; otherwise, if an end-of-file is encountered and no characters have been read into the array or if a read error occurs, then `gets` returns the null pointer.

7.27 Routine putc

Include: `#include <stdio.h>`
Prototype: `int putc(int c, FILE *stream);`

Behavior: `putc` is similar to `fputc` except that it may be implemented as a macro.

7.28 Function putchar

Include: `#include <stdio.h>`
Prototype: `int putchar(int c);`

Behavior: `putchar` is similar to `putc` except that it writes to `stdout`.

7.29 Function puts

Include: `#include <stdio.h>`
Prototype: `int puts(const char *s);`

Behavior: `puts` writes the string pointed to by `s` to `stdout` and prints a new-line character instead of the terminating null character. If successful, `puts` returns a nonnegative value; otherwise, it returns `EOF`.

7.30 Function ungetc

Include: `#include <stdio.h>`
Prototype: `int ungetc(int c, FILE *stream);`

Behavior: `ungetc` pushes argument `c` back into the input stream pointed to by `stream`. Character `c` will be returned by a subsequent read on `stream`. Only one character pushback is guaranteed. The pushed-back character will be discarded if a file positioning function is called with `stream` as an argument.

If successful, `ungetc` clears the end-of-file indicator associated with the stream and returns the pushed-back character `c`; otherwise, it returns `EOF`.

7.31 Function fread

Include: `#include <stdio.h>`
Prototype:
```
size_t fread(void *ptr, size_t size,
             size_t nelem, FILE *stream);
```

Behavior: `fread` reads, into the array pointed to by `ptr`, up to `nelem` elements of size `size` from the stream pointed to by `stream`. `fread` returns the number of elements read successfully.

7.32 Function fwrite

Include: `#include <stdio.h>`
Prototype:
```
size_t fwrite(const void *ptr,
              size_t size, size_t nelem,
              FILE *stream);
```

Behavior: `fwrite` writes, from the array pointed to by `ptr`, up to `nelem` elements of size `size` to the stream pointed to by `stream`. `fwrite` returns the number of elements successfully written.

7.33 Function fgetpos

Include: `#include <stdio.h>`
Prototype: `int fgetpos(FILE *stream, fpos_t *pos);`

Behavior: `fgetpos` stores in `*pos` the current value of the file position indicator associated with the stream pointed to by `stream`. If successful, `fgetpos` returns zero; otherwise, it returns a nonzero value.

7.34 Function fseek

Include: `#include <stdio.h>`
Prototype:
```
int fseek(FILE *stream, long int offset,
          int whence);
```

Behavior: `fseek` sets the file position indicator associated with the stream pointed to by `stream`. For a binary stream, the new position (measured in characters) is equal to `offset` plus the position specified by `whence`:

value of whence	specified position
SEEK_SET	beginning of the file
SEEK_CUR	the current position in the file
SEEK_END	end of the file

For a text stream, `offset` must be equal to zero or it must be a value returned by `ftell` and `whence` must be equal to `SEEK_SET`. After calling `fseek` either input or output can be performed on the stream.

If successful, `fseek` clears the end-of-file indicator, undoes the effects of `ungetc` and returns zero; otherwise, it returns a nonzero value.

7.35 Function fsetpos

Include: `#include <stdio.h>`
Prototype:
```
int fsetpos(FILE *stream,
            const fpos_t *pos);
```

Behavior: `fsetpos` sets the file position indicator for the stream pointed to by `stream` to `*pos`; the value of `*pos` must have been obtained by calling `fgetpos` on the same stream. After calling `fsetpos` either input or output can be performed on the stream.

If successful, `fsetpos` clears the end-of-file indicator, undoes the effects of `ungetc` and returns zero; otherwise, it returns a nonzero value.

7.36 Function ftell

Include: `#include <stdio.h>`
Prototype: `long int ftell(FILE *stream);`

Behavior: `ftell` returns the current value of the file position indicator associated with the stream pointed to by `stream`. If unsuccessful, `ftell` returns `-1L`.

7.37 Function rewind

Include: `#include <stdio.h>`
Prototype: `void rewind(FILE *stream);`

Behavior: `rewind` resets, to the beginning of the file, the file position indicator associated with the stream pointed to by `stream`.

7.38 Function clearerr

Include: `#include <stdio.h>`
Prototype: `void clearerr(FILE *stream);`

Behavior: `clearerr` clears the end-of-file and error indicators associated with the stream pointed to by `stream`.

7.39 Function feof

Include: `#include <stdio.h>`
Prototype: `int feof(FILE *stream);`

Behavior: `feof` returns a nonzero value if the end-of-file indicator is set for the stream pointed to by `stream`.

7.40 Function ferror

Include: `#include <stdio.h>`
Prototype: `int ferror(FILE *stream);`

Behavior: `ferror` returns a nonzero value if the error indicator is set for the stream pointed to by `stream`; otherwise, it returns zero.

7.41 Function perror

Include: `#include <stdio.h>`
Prototype: `void perror(const char *s);`

Behavior: `perror` prints, on `stderr`, an error message corresponding to the value of `errno`. This message is prefixed by the string pointed to by `s`.

8. General Utility Routines

Besides other items, header file `stdlib.h` contains definitions of the following types and macros:

types	macros
`size_t`	`NULL`
`wchar_t`	`EXIT_FAILURE`
`div_t`	`EXIT_SUCCESS`
`ldiv_t`	`RAND_MAX`
	`MB_CUR_MAX`
	`MB_LEN_MAX`

8.1 Function atof

Include: `#include <stdlib.h>`
Prototype: `double atof(const char *nptr);`

Behavior: `atof` converts the string pointed to by `nptr` to a `double` which it returns as its result.

8.2 Function atoi

Include: `#include <stdlib.h>`
Prototype: `int atoi(const char *nptr);`

Behavior: `atoi` converts the initial portion of the string pointed to by `nptr` to an `int` which it returns as its result.

8.3 Function atol

Include: `#include <stdlib.h>`
Prototype: `long int atol(const char *nptr);`

Behavior: `atol` converts the initial portion of the string pointed to by `nptr` to a `long int` which it returns as its result.

8.4 Function strtod

Include: `#include <stdlib.h>`
Prototype:
```
double strtod(const char *nptr,
              char **endptr);
```

Behavior: `strtod` converts the initial portion of the string pointed to by `nptr` to a `double` and returns this real as its result. A pointer to the remaining substring is stored in the object pointed to by `endptr`. In case no conversion is possible, `nptr` is stored in `endptr`. (`endptr` is assigned a value only if it is not the null pointer).

8.5 Function strtol

Include: `#include <stdlib.h>`
Prototype:
```
long int strtol(const char *nptr,
                char **endptr, int base);
```
Behavior: `strtol` converts the initial portion of the string pointed to by `nptr` to a `long int` which it returns as its result. A pointer to the remaining substring is stored in the object pointed to by `endptr`. In case no conversion is possible, `nptr` is stored in `endptr`. (`endptr` is assigned a value only if it is not the null pointer). If `base` is zero, then the string pointed to by `nptr` must be an optionally signed integer constant. For more details about values allowed for `base`, see the *ANSI C Reference Manual* [ANSI C 1988a].

8.6 Function strtoul

Include: `#include <stdlib.h>`
Prototype:
```
unsigned long int strtoul(const char *nptr,
                          char **endptr, int base);
```
Behavior: `strtoul` converts the initial portion of the string pointed to by `nptr` to an `unsigned long int`; this integer is returned as the result. A pointer to the remaining substring is stored in the object pointed to by `endptr`. In case no conversion is possible, `nptr` is stored in `endptr`. (`endptr` is assigned a value only if it is not the null pointer). If `base` is zero, then the string pointed to by `nptr` must be an optionally signed integer constant. For more details about values allowed for `base`, see the *ANSI C Reference Manual* [ANSI C 1988a].

8.7 Function rand

Include: `#include <stdlib.h>`
Prototype: `int rand(void);`
Behavior: `rand` returns a pseudo-random integer between 0 and `RAND_MAX`.

8.8 Function srand

Include: `#include <stdlib.h>`
Prototype: `void srand(unsigned int seed);`
Behavior: `srand` uses the value of `seed` to initiate a new sequence of pseudo-random numbers to be generated by `rand`. Calling `srand` with the same seed value leads to the generation of the same pseudo-random number sequence. The default seed used is one.

8.9 Function calloc

Include: `#include <stdlib.h>`
Prototype: `void *calloc(size_t nelem, size_t size);`
Behavior: `calloc` allocates storage for an array of `nelem` elements, each of size `size`. All bits of the allocated storage are set to zero. If successful, `calloc` returns a pointer to the allocated storage; otherwise, it returns the null pointer.

8.10 Function free

Include: `#include <stdlib.h>`
Prototype: `void free(void *ptr);`
Behavior: `free` deallocates the storage pointed to by `ptr`. The storage pointed to by `ptr` must have been allocated previously by calling `calloc`, `malloc` or `realloc`.

8.11 Function malloc

Include: `#include <stdlib.h>`
Prototype: `void *malloc(size_t size);`
Behavior: `malloc` allocates `size` bytes of storage. If successful, `malloc` returns a pointer to the allocated storage; otherwise, it returns the null pointer.

8.12 Function realloc

Include: `#include <stdlib.h>`
Prototype: `void *realloc(void *ptr, size_t size);`
Behavior: `realloc` changes the size of the object pointed to by `ptr` to `size`. The contents of the object are unchanged (up to the smaller of the new and old sizes); if necessary, the contents of the old storage are copied to the new storage.

If successful, `realloc` returns a pointer to the possibly new allocated space; otherwise, it returns the null pointer (the contents of the old storage are not changed).

8.13 Function abort

Include: `#include <stdlib.h>`
Prototype: `void abort(void);`
Behavior: `abort` causes abnormal termination of the program executing it unless there is a handler for the signal `SIGABRT` (generated by `abort`) and this handler does not return.

8.14 Function atexit

Include: `#include <stdlib.h>`
Prototype: `int atexit(void (*func)(void));`

Behavior: `atexit` registers the function pointed to by `func`, for calling (without arguments) at normal program termination. If `atexit` is successful, then it returns zero; otherwise, it returns a nonzero value.

8.15 Function exit

Include: `#include <stdlib.h>`
Prototype: `void exit(int status);`

Behavior: `exit` causes normal program termination. Prior to program termination, functions registered by calling `atexit` are called, in the reverse order of their registration (a function registered *n* times will be called *n* times.) All open output streams are flushed, all open streams are closed and all temporary files (created by calling `tmpfile`) are removed.

Successful program termination is indicated to the host environment by calling `exit` with the value zero or `EXIT_SUCCESS`; failure is indicated by calling `exit` with the value `EXIT_FAILURE`.

8.16 Function getenv

Include: `#include <stdlib.h>`
Prototype: `char *getenv(const char *name);`

Behavior: `getenv` searches an environment list variable for a string that matches the string pointed to by `name`. If successful, `getenv` returns a pointer to a string associated with the matched string; otherwise, it returns the null pointer.

8.17 Function system

Include: `#include <stdlib.h>`
Prototype: `int system(const char *string);`

Behavior: `system` passes the string pointed to by `string` to the host environment for execution. The value returned by `system` is implementation dependent.

8.18 Function bsearch

Include: `#include <stdlib.h>`
Prototype:
```
void *bsearch(const void *key,
    const void *base, size_t nelem, size_t size,
    int (*cmp)(const void *, const void *));
```

Behavior: `bsearch` searches an array of `nelem` elements, each of size `size`, for an element equal to `*key`; `base` points to the first element of this array which must be sorted in ascending order according to the comparison function `cmp`. This function takes two arguments and returns an integer less than, equal to or greater than zero depending upon whether its first argument is less than, equal to or greater than its second argument.

If successful, `bsearch` returns a pointer to the array element that matches `*key`; otherwise, it returns the null pointer.

8.19 Function qsort

Include: `#include <stdlib.h>`
Prototype:
```
void qsort(void *base, size_t nelem,
    size_t size,
    int (*cmp)(const void *, const void *));
```

Behavior: `qsort` sorts an array of `nelem` elements, each of size `size`; `base` points to the first element of the array to be sorted. The array is sorted in increasing order using the comparison function pointed to by `cmp`, which is called with pointers to the two arguments to be compared. `cmp` returns an integer less than, equal to or greater than zero depending upon whether its first argument is less than, equal to or greater than its second argument.

8.20 Function abs

Include: `#include <stdlib.h>`
Prototype: `int abs(int j);`

Behavior: `abs` returns the absolute value of its argument.

8.21 Function div

Include: `#include <stdlib.h>`
Prototype: `div_t div(int numer, int denom);`

Behavior: `div` returns a structure of type `div_t` that contains the quotient and remainder resulting from dividing `numer` by `denom`:

```
typedef                 struct div_t {
    int quot;    /*quotient*/
    int rem;     /*remainder*/
} div_t;
```

Note that `quot*denom+rem` is equal to `numer`.

8.22 Function labs

Include: `#include <stdlib.h>`
Prototype: `long int labs(long int j);`

Behavior: `labs` is similar to `abs` except it returns a `long int` value.

8.23 Function ldiv

Include: `#include <stdlib.h>`
Prototype:
```
ldiv_t ldiv(long int numer, long int denom);
```
Behavior: `ldiv` is similar to `div` except that the type of its arguments and that of the elements of the structure returned is `long int`.

9. String Handling Routines

Besides other items, header file `string.h` contains the declaration of the type `size_t` and the definition of the macro `NULL`.

9.1 Function memcpy

Include: `#include <string.h>`
Prototype:
```
void *memcpy(void *s1, const void *s2,
                                    size_t n);
```
Behavior: `memcpy` copies `n` characters from the object pointed to by `s2` to the object pointed to by `s1`. Objects pointed to by `s1` and `s2` must not overlap. `memcpy` returns `s1`.

9.2 Function memmove

Include: `#include <string.h>`
Prototype:
```
void *memmove(void *s1, const void *s2,
                                    size_t n);
```
Behavior: `memmove` copies `n` characters from the object pointed to by `s2` to the object pointed to by `s1`. Objects pointed to by `s1` and `s2` can overlap. `memmove` returns `s1`.

9.3 Function strcpy

Include: `#include <string.h>`
Prototype:
```
char *strcpy(char *s1, const char *s2);
```
Behavior: `strcpy` copies the string pointed to by `s2` (including the terminating null character) to the array pointed to by `s1`. Objects pointed to by `s1` and `s2` must not overlap. `strcpy` returns `s1`.

9.4 Function strncpy

Include: `#include <string.h>`
Prototype:
```
char *strncpy(char *s1, const char *s2,
                                    size_t n);
```
Behavior: `strncpy` copies up to `n` characters or up to the null character from the array pointed to by `s2` to the array pointed to by `s1`. These two arrays must not overlap. If the length of the string pointed to by `s2` is less than `n`, then `s1` will be padded with null characters until `n` characters have been written. `strncpy` returns `s1`.

9.5 Function strcat

Include: `#include <string.h>`
Prototype:
```
char *strcat(char *s1, const char *s2);
```
Behavior: `strcat` appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The null character at the end of the string pointed to by `s1` is overwritten. `s1` and `s2` must not overlap. `strcat` returns `s1`.

9.6 Function strncat

Include: `#include <string.h>`
Prototype:
```
char *strncat(char *s1, const char *s2,
                                    size_t n);
```
Behavior: `strncat` appends up to `n` characters or up to the null character from the array pointed to by `s2` to the end of the string pointed to by `s1`. The null character at the end of `s1` is overwritten. A terminating null character is appended to the string pointed to by `s1`. `s1` and `s2` must not overlap. `strncat` returns `s1`.

9.7 Function memcmp

Include: `#include <string.h>`
Prototype:
```
int memcmp(const void *s1, const void *s2,
                                    size_t n);
```
Behavior: `memcmp` compares the first `n` characters of the objects pointed to by `s1` and `s2` and returns an integer greater than, equal to or less than zero, depending upon whether the object pointed to by `s1` is greater than, equal to or less than the object pointed to by `s2`.

9.8 Function strcmp

Include: `#include <string.h>`
Prototype:
```
int strcmp(const char *s1, const char *s2);
```
Behavior: `strcmp` returns an integer greater than, equal to or less than zero, depending upon whether the string pointed to by `s1` is greater than, equal to or less than the string pointed to by `s2`.

9.9 Function strcoll

Include: `#include <string.h>`
Prototype: `int strcoll(const char *s1, const char *s2);`

Behavior: `strcoll` is the same as `strcmp` but the comparison is based on interpreting the strings to be compared according to local conventions.

9.10 Function strncmp

Include: `#include <string.h>`
Prototype: `int strncmp(const char *s1, const char *s2, size_t n);`

Behavior: `strncmp` compares up to `n` characters or up to the null character from the arrays pointed to by `s1` and `s2`.

`strncmp` returns an integer greater than, equal to or less than zero, depending upon whether the array pointed to by `s1` is greater than, equal to or less than the array pointed to by `s2`.

9.11 Function strxfrm

Include: `#include <string.h>`
Prototype: `size_t strxfrm(char *s1, const char *s2, size_t n);`

Behavior: `strxfrm` transforms up to `n` characters (including the terminating null character) of the string pointed to by `s2` (as described below) and places the resulting string in the array pointed to by `s1`. Objects pointed to by `s1` and `s2` must not overlap. The string pointed to by `s2` is transformed so that the result of comparing two transformed strings with `strcmp` is equal to the result of comparing the two original strings with `strcoll`.

`strxfrm` returns the length of the transformed string.

9.12 Function memchr

Include: `#include <string.h>`
Prototype: `void *memchr(const void *s, int c, size_t n);`

Behavior: `memchr` returns a pointer to the first occurrence of `c` (converted to an `unsigned char`) in the first `n` characters of the string pointed to by `s`. If `memchr` does not find such a `c`, then it returns the null pointer.

9.13 Function strchr

Include: `#include <string.h>`
Prototype: `char *strchr(const char *s, int c);`

Behavior: `strchr` returns a pointer to the first occurrence of `c` (converted to `char`) in the string pointed to by `s` (the terminating null character is also considered to be part of the string). If `strchr` does not find such a `c`, then it returns the null pointer.

9.14 Function strcspn

Include: `#include <string.h>`
Prototype: `size_t strcspn(const char *s1 const char *s2);`

Behavior: `strcspn` returns the length of the maximum prefix of string pointed to by `s1`, which consists of characters *not* in the string pointed to by `s2`.

9.15 Function strpbrk

Include: `#include <string.h>`
Prototype: `char *strpbrk(const char *s1, const char *s2);`

Behavior: `strpbrk` returns a pointer to the first occurrence of any character from the string pointed to by `s2` in the string pointed to by `s1`. If there is no such character, then `strpbrk` returns the null pointer.

9.16 Function strrchr

Include: `#include <string.h>`
Prototype: `char *strrchr(const char *s, int c);`

Behavior: `strrchr` returns a pointer to the last occurrence of `c` (converted to `char`) in the string pointed to by `s` (the terminating null character is also considered to be part of the string). If `c` does not occur in `s`, then `strrchr` returns the null pointer.

9.17 Function strspn

Include: `#include <string.h>`
Prototype: `size_t strspn(const char *s1, const char *s2);`

Behavior: `strspn` returns the length of the maximum prefix of the string pointed to by `s1` which consists of just the characters in the string pointed to by `s2`.

9.18 Function strstr

Include: `#include <string.h>`
Prototype: `char *strstr(const char *s1, const char *s2);`

Behavior: `strstr` returns a pointer to the first substring in the string pointed to by `s1` that matches the string pointed to by `s2`. If there is no such substring, `strstr` returns the null pointer. If `s2` points to a zero-length string, then `strstr` returns `s1`.

9.19 Function strtok

Include: `#include <string.h>`
Prototype: `char *strtok(char *s1, const char *s2);`

Behavior: A series of calls to `strtok` splits the string pointed to by `s1` into a series of tokens (items), each of which is delimited by a character from the separator string pointed to by `s2`. The first argument of the first call is the string to be split into tokens; this argument is replaced in subsequent calls by the null pointer. Leading occurrences of characters from the separator string pointed to by `s2` in the string pointed to by `s1` are ignored. The separator string can vary from call to call.

If a token is found, then `strtok` returns a pointer to the first character of the token; otherwise, it returns the null pointer.

9.20 Function memset

Include: `#include <string.h>`
Prototype: `void *memset(void *s, int c, size_t n);`

Behavior: `memset` sets each of the first `n` characters of the object pointed to by `s` to the character `c`. `memset` returns `s`.

9.21 Function strlen

Include: `#include <string.h>`
Prototype: `size_t strlen(const char *s);`

Behavior: `strlen` returns the length of the string pointed to by `s` (excluding the terminating null character).

10. Date and Time Functions

Besides other items, header file `time.h` contains the definition of the macros `NULL` and the `CLK_TCK`, the declarations of the types `size_t`, `clock_t`, `time_t` and `struct tm`.

`clock_t` and `time_t` are arithmetic types capable of representing times. Structure `tm` must have at least the following components:


```
int tm_sec;   /*seconds: 0 to 59*/
int tm_min;   /*minutes: 0 to 59*/
int tm_hour;  /*hours: 0 to 23*/
int tm_mday;  /*day: 1 to 31*/
int tm_mon;   /*month: 0 to 11*/
int tm_year;  /*years since 1900*/
int tm_wday;  /*days since Sunday: 0 to 6*/
int tm_yday;  /*days since January 1: 0 to 365*/
int tm_isdst; /*Daylight Saving Time flag*/
```

`tm_isdst` is positive if Daylight Saving Time is in effect, zero if it is not in effect, and negative if information is unavailable.

10.1 Function clock

Include: `#include <time.h>`
Prototype: `clock_t clock(void);`

Behavior: `clock` returns the processor time in clock ticks used since the beginning of program execution. To determine the time in seconds, the value returned by `clock` is divided by `CLK_TCK`.

10.2 Function difftime

Include: `#include <time.h>`
Prototype: `double difftime(time_t t1, time_t t0);`

Behavior: `difftime` returns the value of the expression `t1-t0` (in seconds).

10.3 Function mktime

Include: `#include <time.h>`
Prototype: `time_t mktime(struct tm *timeptr);`

Behavior: `mktime` returns the calendar time (using the encoding used by `time`) corresponding to the time specified as components of the structure pointed to by `timeptr`.

10.4 Function time

Include: `#include <time.h>`
Prototype: `time_t time(time_t *timer);`

Behavior: `time` returns the current calendar time. If `timer` is not equal to the null pointer, then the value returned is stored in `*timer`.

10.5 Function asctime

Include: `#include <time.h>`
Prototype: `char *asctime(const struct tm *timeptr);`

Behavior: `asctime` converts the time specified as components in the structure `*timeptr` into a string of the form

```
Wed May 18 22:43:56 1988\n\0
```

and returns a pointer to this string.

10.6 Function ctime

Include: `#include <time.h>`
Prototype: `char *ctime(const time_t *timer);`

Behavior: `ctime` converts the calendar time pointed to by `timer` to local time and returns a pointer to the string containing the local time.

10.7 Function gmtime

Include: `#include <time.h>`
Prototype: `struct tm *gmtime(const time_t *timer);`

Behavior: `gmtime` splits the calendar time pointed to by `timer` into components in terms of the Coordinated Universal Time (UTC) and returns a pointer to the structure containing the components.

10.8 Function localtime

Include: `#include <time.h>`
Prototype: `struct tm *localtime(const time_t *timer);`

Behavior: `localtime` splits the calendar time pointed to by `timer` into components expressed as local time. It returns a pointer to a structure containing the components.

Appendix D

Differences between ANSI C and K&R C

I will now summarize the important differences between K&R C and ANSI C [Relph 1987; Kernighan & Ritchie 1978; ANSI C 1988a]. Unless qualified by K&R C or ANSI C, the discussion refers to the changes made to K&R C by the ANSI C standardization process, i.e., the discussion refers to ANSI C specific facilities. Note that many C compilers already implement some ANSI C extensions. This is because these compilers implement an extended form of K&R C that is described in *The C Programming Language—Reference Manual* [Ritchie 1980]. Extended K&R C includes features such as enumeration and `void` types, structure arguments in function calls, and structure assignment; these features have been incorporated into ANSI C.

1. General

1. A standard character set is specified whereas in K&R C the character set was implementation dependent.
2. The following trigraph sequences, denoting characters not found on all keyboards, are supported:

trigraph sequence	character denoted
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

3. Keywords `const`, `volatile`, `enum`, `signed` and `void` have been added.
4. Keywords `entry`, `fortran` and `asm` have been deleted.

APPENDIX E

Operator Precedence and Associativity

Operators	*Associativity*
`()` `[]` `->` `.`	left to right
`++` `--` `!` `~` `sizeof` *(type)* `+` *(unary)* `-` *(unary)* `*` *(indirection)* `&` *(address)*	right to left
`*` `/` `%`	left to right
`+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=` `+=` `-=` `*=` `/=` *etc.*	right to left
`,` *(comma operator)*	left to right

GDB QUICK REFERENCE GDB Version 4

Essential Commands

`gdb` *program* [*core*]	debug *program* [using coredump *core*]
`b` [*file*`:`]*function*	set breakpoint at *function* [in *file*]
`run` [*arglist*]	start your program [with *arglist*]
`bt`	backtrace: display program stack
`p` *expr*	display the value of an expression
`c`	continue running your program
`n`	next line, stepping over function calls
`s`	next line, stepping into function calls

Starting GDB

`gdb`	start GDB, with no debugging files
`gdb` *program*	begin debugging *program*
`gdb` *program core*	debug coredump *core* produced by *program*
`gdb --help`	describe command line options

Stopping GDB

`quit`	exit GDB; also `q` or `EOF` (eg `C-d`)
`INTERRUPT`	(eg `C-c`) terminate current command, or send to running process

Getting Help

`help`	list classes of commands
`help` *class*	one-line descriptions for commands in *class*
`help` *command*	describe *command*

Executing your Program

`run` *arglist*	start your program with *arglist*
`run`	start your program with current argument list
`run ...` <*inf* >*outf*	start your program with input, output redirected
`kill`	kill running program
`tty` *dev*	use *dev* as stdin and stdout for next `run`
`set args` *arglist*	specify *arglist* for next `run`
`set args`	specify empty argument list
`show args`	display argument list
`show env`	show all environment variables
`show env` *var*	show value of environment variable *var*
`set env` *var string*	set environment variable *var*
`unset env` *var*	remove *var* from environment

Shell Commands

`cd` *dir*	change working directory to *dir*
`pwd`	Print working directory
`make ...`	call "`make`"
`shell` *cmd*	execute arbitrary shell command string

[ ] surround optional ... to mean more ...

Breakpoints and Watchpoints

`break` [*file*`:`]*line* `b` [*file*`:`]*line*	set breakpoint at *line* number [in *file*] eg: `break main.c:37`
`break` [*file*`:`]*func*	set breakpoint at *func* [in *file*]
`break +`*offset*	set break at *offset* lines from current stop
`break -`*offset*	
`break *`*addr*	set breakpoint at address *addr*
`break`	set breakpoint at next instruction
`break ... if` *expr*	break conditionally on nonzero *expr*
`cond` *n* [*expr*]	new conditional expression on breakpoint *n*; make unconditional if no *expr*
`tbreak ...`	temporary break; disable when reached
`rbreak` *regex*	break on all functions matching *regex*
`watch` *expr*	set a watchpoint for expression *expr*
`catch` *x*	break at C++ handler for exception *x*
`info break`	show defined breakpoints
`info watch`	show defined watchpoints
`clear`	delete breakpoints at next instruction
`clear` [*file*`:`]*fun*	delete breakpoints at entry to *fun*()
`clear` [*file*`:`]*line*	delete breakpoints on source line
`delete` [*n*]	delete breakpoints [or breakpoint *n*]
`disable` [*n*]	disable breakpoints [or breakpoint *n*]
`enable` [*n*]	enable breakpoints [or breakpoint *n*]
`enable once` [*n*]	enable breakpoints [or breakpoint *n*]; disable again when reached
`enable del` [*n*]	enable breakpoints [or breakpoint *n*]; delete when reached
`ignore` *n count*	ignore breakpoint *n*, *count* times
`commands` *n* [`silent`] *command-list*	execute GDB *command-list* every time breakpoint *n* is reached. [`silent` suppresses default display]
`end`	end of *command-list*

Program Stack

`backtrace` [*n*] `bt` [*n*]	print trace of all frames in stack; or of *n* frames—innermost if *n*>0, outermost if *n*<0
`frame` [*n*]	select frame number *n* or frame at address *n*; if no *n*, display current frame
`up` *n*	select frame *n* frames up
`down` *n*	select frame *n* frames down
`info frame` [*addr*]	describe selected frame, or frame at *addr*
`info args`	arguments of selected frame
`info locals`	local variables of selected frame
`info reg` [*rn*]...	register values [for regs *rn*] in selected frame; `all-reg` includes floating point
`info all-reg` [*rn*]	
`info catch`	exception handlers active in selected frame

Execution Control

`continue` [*count*] `c` [*count*]	continue running; if *count* specified, ignore this breakpoint next *count* times
`step` [*count*] `s` [*count*]	execute until another line reached; repeat *count* times if specified
`stepi` [*count*] `si` [*count*]	step by machine instructions rather than source lines
`next` [*count*] `n` [*count*]	execute next line, including any function calls
`nexti` [*count*] `ni` [*count*]	next machine instruction rather than source line
`until` [*location*]	run until next instruction (or *location*)
`finish`	run until selected stack frame returns
`return` [*expr*]	pop selected stack frame without executing [setting return value]
`signal` *num*	resume execution with signal *s* (none if 0)
`jump` *line*	resume execution at specified *line* number
`jump *`*address*	or *address*
`set var=`*expr*	evaluate *expr* without displaying it; use for altering program variables

Display

`print` [*/f*] [*expr*] `p` [*/f*] [*expr*]	show value of *expr* [or last value `$`] according to format *f*:
`x`	hexadecimal
`d`	signed decimal
`u`	unsigned decimal
`o`	octal
`t`	binary
`a`	address, absolute and relative
`c`	character
`f`	floating point
`call` [*/f*] *expr*	like `print` but does not display `void`
`x` [*/Nuf*] *expr*	examine memory at address *expr*; optional format spec follows slash
N	count of how many units to display
u	unit size; one of `b` individual bytes `h` halfwords (two bytes) `w` words (four bytes) `g` giant words (eight bytes)
f	printing format. Any `print` format, or `s` null-terminated string `i` machine instructions
`disassem` [*addr*]	display memory as machine instructions

Automatic Display

`display` [*/f*] *expr*	show value of *expr* each time program stops [according to format *f*]
`display`	display all enabled expressions on list
`undisplay` *n*	remove number(s) *n* from list of automatically displayed expressions
`disable disp` *n*	disable display for expression(s) number *n*

Expressions

expr	an expression in C, C++, or Modula-2 (including function calls), or:
addr@*len*	an array of *len* elements beginning at *addr*
file::*nm*	a variable or function *nm* defined in *file*
{*type*}*addr*	read memory at *addr* as specified *type*
$	most recent displayed value
$*n*	*n*th displayed value
$$	displayed value previous to $
$$*n*	*n*th displayed value back from $
$_	last address examined with x
$__	value at address $_
$*var*	convenience variable; assign any value
show values [*n*]	show last 10 values [or surrounding $*n*]
show conv	display all convenience variables

Symbol Table

info address *s*	show where symbol *s* is stored
info func [*regex*]	show names, types of defined functions (all, or matching *regex*)
info var [*regex*]	show names, types of global variables (all, or matching *regex*)
whatis [*expr*]	show data type of *expr* [or $] without evaluating; **ptype** gives more detail
ptype [*expr*]	
ptype *type*	describe type, struct, union, or enum

GDB Scripts

source *script*	read, execute GDB commands from file *script*
define *cmd*	create new GDB command *cmd*;
command-list	execute script defined by *command-list*
end	end of *command-list*
document *cmd*	create online documentation for new GDB
help-text	command *cmd*
end	end of *help-text*

Signals

handle *signal act*	specify GDB actions for *signal*:
print	announce signal
noprint	be silent for signal
stop	halt execution on signal
nostop	do not halt execution
pass	allow your program to handle signal
nopass	do not allow your program to see signal
info signals	show table of signals, GDB action for each

Debugging Targets

target *type param*	connect to target machine, process, or file
help target	display available targets
attach *param*	connect to another process
detach	release target from GDB control

Controlling GDB

set *param value*	set one of GDB's internal parameters
show *param*	display current setting of parameter

Parameters understood by **set** and **show**:

complaint *limit*	number of messages on unusual symbols
confirm *on/off*	enable or disable cautionary queries
editing *on/off*	control **readline** command-line editing
height *lpp*	number of lines before pause in display
language *lang*	Language for GDB expressions (**auto**, **c** or **modula-2**)
listsize *n*	number of lines shown by **list**
prompt *str*	use *str* as GDB prompt
radix *base*	octal, decimal, or hex number representation
verbose *on/off*	control messages when loading symbols
width *cpl*	number of characters before line folded
write *on/off*	Allow or forbid patching binary, core files (when reopened with **exec** or **core**)
history ...	groups with the following options:
h ...	
h exp *off/on*	disable/enable **readline** history expansion
h file *filename*	file for recording GDB command history
h size *size*	number of commands kept in history list
h save *off/on*	control use of external file for command history
print ...	groups with the following options:
p ...	
p address *on/off*	print memory addresses in stacks, values
p array *off/on*	compact or attractive format for arrays
p demangl *on/off*	source (demangled) or internal form for C++ symbols
p asm-dem *on/off*	demangle C++ symbols in machine-instruction output
p elements *limit*	number of array elements to display
p object *on/off*	print C++ derived types for objects
p pretty *off/on*	struct display: compact or indented
p union *on/off*	display of union members
p vtbl *off/on*	display of C++ virtual function tables
show commands	show last 10 commands
show commands *n*	show 10 commands around number *n*
show commands +	show next 10 commands

Working Files

file [*file*]	use *file* for both symbols and executable; with no arg, discard both
core [*file*]	read *file* as coredump; or discard
exec [*file*]	use *file* as executable only; or discard
symbol [*file*]	use symbol table from *file*; or discard
load *file*	dynamically link *file* and add its symbols
add-sym *file addr*	read additional symbols from *file*, dynamically loaded at *addr*
info files	display working files and targets in use
path *dirs*	add *dirs* to front of path searched for executable and symbol files
show path	display executable and symbol file path
info share	list names of shared libraries currently

Source Files

dir *names*	add directory *names* to front of source path
dir	clear source path
show dir	show current source path
list	show next ten lines of source
list -	show previous ten lines
list *lines*	display source surrounding *lines*, specified as:
[*file*:]*num*	line number [in named file]
[*file*:]*function*	beginning of function [in named file]
+*off*	*off* lines after last printed
-*off*	*off* lines previous to last printed
***address*	line containing *address*
list *f*,*l*	from line *f* to line *l*
info line *num*	show starting, ending addresses of compiled code for source line *num*
info source	show name of current source file
info sources	list all source files in use
forw *regex*	search following source lines for *regex*
rev *regex*	search preceding source lines for *regex*

GDB under GNU Emacs

M-x gdb	run GDB under Emacs
C-h m	describe GDB mode
M-s	step one line (**step**)
M-n	next line (**next**)
M-i	step one instruction (**stepi**)
C-c C-f	finish current stack frame (**finish**)
M-c	continue (**cont**)
M-u	up *arg* frames (**up**)
M-d	down *arg* frames (**down**)
C-x &	copy number from point, insert at end
C-x SPC	(in source file) set break at point

GDB License

show copying	Display GNU General Public License
show warranty	There is NO WARRANTY for GDB. Display full no-warranty statement.