The C Library Reference Guide

by Eric Huss

This book can be found at the following address

http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

Introduction

Welcome to the C Library Reference Guide. This guide provides a useful look at the standard C programming language. In no way does this guide attempt to teach one how to program in C, nor will it attempt to provide the history of C or the various implementations of it. It is merely a handy reference to the standard C library. This guide is not a definitive look at the entire ANSI C standard. Some outdated information has been left out. It is simply a quick reference to the functions and syntax of the language. All efforts have been taken to make sure the information contained herein is correct, but no guarantees are made. Nearly all of the information was obtained from the official ANSI C Standard published in 1989 in the document ANSI X3.159-1989. The associated International Organization for Standardization document, ISO 9899-1990, is a near duplicate of the ANSI standard.

This guide is divided into two sections. The first part, "Language", is an analysis of the syntax and the environment. The second part, "Library", is a list of the functions available in the standard C library. These parts were designed to insure conformity among various implementations of the C language. Not all information from the ANSI standard is contained in this guide. Additional reference may be made to the actual ANSI publication.

Return to the Index

The C Library Reference Guide

by Eric Huss

Release 1

Questions, comments, or error reports? Please send them to Eric Huss

The C Library Reference Guide

Index

[# | __ | _ | A | B | C | D | E | F | G | H | I | J | L | M | N | O | P | Q | R | S | T | U | V | W]
Table of Contents

Previous Section | Table of Contents | Index | Next Section 1.2 Identifiers
TOC

1.1.1 Trigraph Characters

A trigraph sequence found in the source code is converted to its respective translation character. This allows people to enter certain characters that are not allowed under some (rare) platforms.

Trigraph Sequence	Translation Character
??=	**#**
??(	**[**
??/	****
??)	**]**
??'	**^**
??<	**{**
??!	**|**
??>	**}**
??-	**~**

Example:

```
printf("No???/n");
```

translates into:

```
printf("No?\n");
```

1.1.2 Escape sequences

The following escape sequences allow special characters to be put into the source code.

Escape Sequence	Name	Meaning
\a	Alert	Produces an audible or visible alert.
\b	Backspace	Moves the cursor back one position (non-destructive).
\f	Form Feed	Moves the cursor to the first position of the next page.

\n	New Line	Moves the cursor to the first position of the next line.
\r	Carriage Return	Moves the cursor to the first position of the current line.
\t	Horizontal Tab	Moves the cursor to the next horizontal tabular position.
\v	Vertical Tab	Moves the cursor to the next vertical tabular position.
\'		Produces a single quote.
\"		Produces a double quote.
\?		Produces a question mark.
****		Produces a single backslash.
\0		Produces a null character.
*****ddd*		Defines one character by the octal digits (base-8 number). Multiple characters may be defined in the same escape sequence, but the value is implementation-specific (see examples).
\x*dd*		Defines one character by the hexadecimal digit (base-16 number).

Examples:

printf("\12");
Produces the decimal character 10 (x0A Hex).

printf("\xFF");
Produces the decimal character -1 or 255 (depending on sign).

printf("\x123");
Produces a single character (value is undefined). May cause errors.

printf("\0222");
Produces two characters whose values are implementation-specific.

1.1.3 Comments

Comments in the source code are ignored by the compiler. They are encapsulated starting with **/*** and ending with ***/**. According to the ANSI standard, nested comments are not allowed, although some implementations allow it.

Single line comments are becoming more common, although not defined in the ANSI standard. Single line comments begin with **//** and are automatically terminated at the end of the current line.

Previous Section | Table of Contents | Index | Next Section 1.2 Identifiers
TOC

Previous Section 1.1 Characters | Table of Contents | Index | Next Section 1.3 Functions

1.2.1 Keywords

The following keywords are reserved and may not be used as an identifier for any other purpose.

```
auto     double int      long
break    else   long     switch
case     enum   register typedef
char     extern return   union
const    float  short    unsigned
continue for    signed   void
default  goto   sizeof   volatile
do       if     static   while
```

1.2.2 Variables

A variable may be defined using any uppercase or lowercase character, a numerical digit (0 through 9), and the underscore character (_). The first character of the variable may not be a numerical digit or underscore. Variable names are case sensitive.

The scope of the variable (where it can be used), is determined by where it is defined. If it is defined outside any block or list of parameters, then it has *file scope*. This means it may be accessed anywhere in the current source code file. This is normally called a global variable and is normally defined at the top of the source code. All other types of variables are local variables. If a variable is defined in a block (encapsulated with **{** and **}**), then its scope begins when the variable is defined and ends when it hits the terminating **}**. This is called *block scope*. If the variable is defined in a function prototype, then the variable may only be accessed in that function. This is called *function prototype scope*.

Access to variables outside of their file scope can be made by using *linkage*. Linkage is done by placing the keyword **extern** prior to a variable declaration. This allows a variable that is defined in another source code file to be accessed.

Variables defined within a function scope have *automatic storage duration*. The life of the variable is determined by the life of the function. Space is allocated at the beginning of the function and terminated

at the end of the function. *Static storage duration* can be obtained by placing the keyword **`static`** in front of the variable declaration. This causes the variable's space to be allocated when the program starts up and is kept during the life of the program. The value of the variable is preserved during subsequent calls to the function that defines it. Variables with file scope are automatically static variables.

A variable is defined by the following:

storage-class-specifier type-specifier variable-names,...

The storage-class-specifier can be one of the following:

`typedef`	The symbol name "*variable-name*" becomes a type-specifier of type "*type-specifier*". No variable is actually created, this is merely for convenience.
`extern`	Indicates that the variable is defined outside of the current file. This brings the variables scope into the current scope. No variable is actually created by this.
`static`	Causes a variable that is defined within a function to be preserved in subsequent calls to the function.
`auto`	Causes a local variable to have a local lifetime (default).
`register`	Requests that the variable be accessed as quickly as possible. This request is not guaranteed. Normally, the variable's value is kept within a CPU register for maximum speed.

The type-specifier can be one of the following:

`void`	Defines an empty or NULL value whose type is incomplete.
`char, signed char`	Variable is large enough to store a basic character in the character set. The value is either signed or nonnegative.
`unsigned char`	Same as char, but unsigned values only.
`short, signed short, short int, signed short int`	Defines a short signed integer. May be the same range as a normal int, or half the bits of a normal int.
`unsigned short, unsigned short int`	Defines an unsigned short integer.
`int, signed, signed int`, or no type specifier	Defines a signed integer. If no type specifier is given, then this is the default.
`unsigned int, unsigned`	Same as int, but unsigned values only.
`long, signed long, long int, signed long int`	Defines a long signed integer. May be twice the bit size as a normal int, or the same as a normal int.

unsigned long, unsigned long int	Same as long, but unsigned values only.
float	A floating-point number. Consists of a sign, a mantissa (number greater than or equal to 1), and an exponent. The mantissa is taken to the power of the exponent then given the sign. The exponent is also signed allowing extremely small fractions. The mantissa gives it a finite precision.
double	A more accurate floating-point number than float. Normally twice as many bits in size.
long double	Increases the size of double.

Here are the maximum and minimum sizes of the type-specifiers on most common implementations. Note: some implementations may be different.

Type	Size	Range
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	1.17549435 * (10^-38) to 3.40282347 * (10^+38)
double	64 bits	2.2250738585072014 * (10^-308) to 1.7976931348623157 * (10^+308)
long double	80 bits	3.4 * (10^-4932) to 1.1 * (10^4932)

Examples:

```
int bob=32;
```
Creates variable "bob" and initializes it to the value 32.

```
char loop1,loop2,loop3='\x41';
```
Creates three variables. The value of "loop1" and "loop2" is undefined. The value of loop3 is the letter "A".

```
typedef char boolean;
```
Causes the keyword "boolean" to represent variable-type "char".

`boolean yes=1;`
Creates variable "yes" as type "char" and sets its value to 1.

1.2.3 Enumerated Tags

Enumeration allows a series of constant integers to be easily assigned. The format to create a enumeration specifier is:

`enum` *identifier* **`{`***enumerator-list***`};`**
Identifier is a handle for identification, and is optional.
Enumerator-list is a list of variables to be created. They will be constant integers. Each variable is given the value of the previous variable plus 1. The first variable is given the value of 0.

Examples:

`enum {joe, mary, bob, fran};`
Creates 4 variables. The value of joe is 0, mary is 1, bob is 2, and fran is 3.

enum test {larry, floyd=20, ted};
Creates 3 variables with the identifier test. The value of larry is 0, floyd is 20, and ted is 21.

1.2.4 Arrays

Arrays create single or multidimensional matrices. They are defined by appending an integer encapsulated in brackets at the end of a variable name. Each additional set of brackets defines an additional dimension to the array. When addressing an index in the array, indexing begins at 0 and ends at 1 less than the defined array. If no initial value is given to the array size, then the size is determined by the initializers. When defining a multidimensional array, nested curly braces can be used to specify which dimension of the array to initialize. The outermost nest of curly braces defines the leftmost dimension, and works from left to right.

Examples:

`int x[5];`
Defines 5 integers starting at x[0], and ending at x[4]. Their values are undefined.

`char str[16]="Blueberry";`
Creates a string. The value at str[8] is the character "y". The value at str[9] is the null

character. The values from str[10] to str[15] are undefined.

char s[]="abc";
Dimensions the array to 4 (just long enough to hold the string plus a null character), and stores the string in the array.

int y[3]={4};
Sets the value of y[0] to 4 and y[1] and y[2] to 0.

```
int joe[4][5]={
     {1,2,3,4,5},
     {6,7,8,9,10},
     {11,12,13,14,15}
};
```

The first row initializes joe[0], the second row joe[1] and so forth. joe[3] is initialized to 5 zeros.

The same effect is achieved by:
int joe[4][5]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

1.2.5 Structures and Unions

Structures and unions provide a way to group common variables together. To define a structure use:

```
struct structure-name {

          variables,...

} structure-variables,...;
```

Structure-name is optional and not needed if the structure variables are defined. Inside it can contain any number of variables separated by semicolons. At the end, *structure-variables* defines the actual names of the individual structures. Multiple structures can be defined by separating the variable names with commas. If no *structure-variables* are given, no variables are created. *Structure-variables* can be defined separately by specifying:

struct *structure-name new-structure-variable*;

new-structure-variable will be created and has a separate instance of all the variables in *structure-name*.

To access a variable in the structure, you must use a record selector (`.`).

Unions work in the same way as structures except that all variables are contained in the same location in memory. Enough space is allocated for only the largest variable in the union. All other variables must share the same memory location. Unions are defined using the union keyword.

Examples:

```
struct my-structure {
     int fred[5];
     char wilma, betty;
     float barny=1;
};
```

This defines the structure my-structure, but nothing has yet been done.

```
struct my-structure account1;
```
This creates account1 and it has all of the variables from my-structure. account1.barny contains the value "1".

```
union my-union {
     char character_num;
     int integer_num;
     long long_num;
     float float_num;
     double double_num;
} number;
```

This defines the union number and allocates just enough space for the variable double_num.

```
number.integer_num=1;
```
Sets the value of integer_num to "1".
```
number.float_num=5;
```
Sets the value of float_num to "5".
```
printf("%i",integer_num);
```
This is undefined since the location of integer_num was overwritten in the previous line by float_num.

1.2.6 Constants

Constants provide a way to define a variable which cannot be modified by any other part in the code.

Constants can be defined by placing the keyword **const** in front of any variable declaration. If the keyword **volatile** is placed after **const**, then this allows external routines to modify the variable (such as hardware devices). This also forces the compiler to retrieve the value of the variable each time it is referenced rather than possibly optimizing it in a register.

Constant numbers can be defined in the following way:

Hexadecimal constant:

0x *hexadecimal digits...*
Where *hexadecimal digits* is any digit or any letter **A** through **F** or **a** through **f**.

Decimal constant:

Any number where the first number is not zero.

Octal constant:

Any number where the first number must be zero.

Floating constant:

A fractional number, optionally followed by either **e** or **E** then the exponent.

The number may be suffixed by:
U or **u**:

Causes the number to be an unsigned long integer.

L or **l**:

If the number is a floating-point number, then it is a long double, otherwise it is an unsigned long integer.

F or **f**:

Causes the number to be a floating-point number.

Examples:

`const float PI=3.141;`
Causes the variable PI to be created with value 3.141. Any subsequent attempts to write to PI are not allowed.

`const int joe=0xFFFF;`
Causes joe to be created with the value of 65535 decimal.

`const float penny=7.4e5;`
Causes penny to be created with the value of 740000.000000.

1.2.7 Strings

Strings are simply an array of characters encapsulated in double quotes. At the end of the string a null character is appended.

Examples:

`"\x65"` and **`"A"`** are the same string.

`char fred[25]="He said, \"Go away!\"";`
The value at fred[9] is a double quote. The value at fred[20] is the null character.

1.2.8 sizeof Keyword

Declaration:

`size_t sizeof` *expression*

or

`size_t sizeof (`*type***`)`**

The sizeof keyword returns the number of bytes of the given expression or type.
`size_t` is an unsigned integer result.

Example:

```
printf("The number of bytes in an int is %d.\n",sizeof
(int));
```

Previous Section
1.1 Characters

| Table of Contents | Index |

Next Section
1.3 Functions

Previous Section 1.2 Identifiers | Table of Contents | Index | Next Section 1.4 References

1.3.1 Function Definition

A function is declared in the following manner:

return-type function-name`(`*parameter-list,...*`) {` *body...* `}`

return-type is the variable type that the function returns. This can not be an array type or a function type. If not given, then **`int`** is assumed.

function-name is the name of the function.

parameter-list is the list of parameters that the function takes separated by commas. If no parameters are given, then the function does not take any and should be defined with an empty set of parenthesis or with the keyword **`void`**. If no variable type is in front of a variable in the paramater list, then **`int`** is assumed. Arrays and functions are not passed to functions, but are automatically converted to pointers. If the list is terminated with an ellipsis (**`,...`**), then there is no set number of parameters. Note: the header **`stdarg.h`** can be used to access arguments when using an ellipsis.

If the function is accessed before it is defined, then it must be prototyped so the compiler knows about the function. Prototyping normally occurs at the beginning of the source code, and is done in the following manner:

return-type function-name`(`*paramater-type-list*`);`

return-type and *function-name* must correspond exactly to the actual function definition. *parameter-type-list* is a list separated by commas of the types of variable parameters. The actual names of the parameters do not have to be given here, although they may for the sake of clarity.

Examples:

```
int joe(float, double, int);
```
This defines the prototype for function joe.

```
int joe(float coin, double total, int sum)
```

```
{
  /*...*/
}
```

This is the actual function joe.

`int mary(void), *lloyd(double);`
This defines the prototype for the function mary with no parameters and return type int. Function llyod is defined with a double type paramater and returns a pointer to an int.

`int (*peter)();`
Defines peter as a pointer to a function with no parameters specified. The value of peter can be changed to represent different functions.

`int (*aaron(char *(*)(void)) (long, int);`
Defines the function aaron which returns a pointer to a function. The function aaron takes one argument: a pointer to a function which returns a character pointer and takes no arguments. The returned function returns a type int and has two parameters of type long and int.

1.3.2 Program Startup

A program begins by calling the function main. There is no prototype required for this. It can be defined with no parameters such as:

`int main(void) {` *body...* **`}`**

Or with the following two parameters:

`int main(int argc, char *argv[]) {` *body...* **`}`**

Note that they do not have to be called **`argc`** or **`argv`**, but this is the common naming system.

`argc` is a nonnegative integer. If **`argc`** is greater than zero, then the string pointed to by **`argv[0]`** is the name of the program. If **`argc`** is greater than one, then the strings pointed to by **`argv[1]`** through **`argv[argc-1]`** are the parameters passed to the program by the system.

Example:

```
#include<stdio.h>

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

```
{
  int loop;

  if(argc>0)
    printf("My program name is %s.\n",argv[0]);

  if(argc>1)
   {
    for(loop=1;loop<argc;loop++)
      printf("Parameter #%i is %s.\n",loop,argv[loop]);
   }
}
```

Previous Section 1.2 Identifiers | Table of Contents | Index | Next Section 1.4 References

Previous Section 1.3 Functions | Table of Contents | Index | Next Section 1.5 Operators

1.4.1 Pointers and the Address Operator

Pointers are variables that contain the memory address for another variable. A pointer is defined like a normal variable, but with an asterisk before the variable name. The type-specifier determines what kind of variable the pointer points to but does not affect the actual pointer.

The address operator causes the memory address for a variable to be returned. It is written with an ampersand sign before the variable name.

When using a pointer, referencing just the pointer such as:

```
int *my_pointer;
int barny;
my_pointer=&barny;
```

Causes my_pointer to contain the address of barny. Now the pointer can be use indirection to reference the variable it points to. Indirection is done by prefixing an asterisk to the pointer variable.

```
*my_pointer=3;
```

This causes the value of barny to be 3. Note that the value of **`my_pointer`** is unchanged.

Pointers offer an additional method for addressing an array. The following array:

```
int my_array[3];
```

Can be addressed normally such as:

```
my_array[2]=3;
```

The same can be accomplished with:

```
*(my_array+2)=3;
```

Note that **`my_array`** is a pointer *constant*. Its value cannot be modified such as:

`my_array++;` This is illegal.

However, if a pointer variable is created such as:

```
int *some_pointer=my_array;
```

Then modifying the pointer will correctly increment the pointer so as to point to the next element in the array.

```
*(some_pointer+1)=3;
```

This will cause the value of **`my_array[1]`** to be 3. On a system where an **`int`** takes up two bytes, adding 1 to **`some_pointer`** did not actually increase it by 1, but by 2 so that it pointed to the next element in the array.

Functions can also be represented with a pointer. A function pointer is defined in the same way as a function prototype, but the function name is replaced by the pointer name prefixed with an asterisk and encapsulated with parenthesis. Such as:

```
int (*fptr)(int, char);
fptr=some_function;
```

To call this function:

```
(*ftpr)(3,'A');
```

This is equivalent to:

```
some_function(3,'A');
```

A structure or union can have a pointer to represent it. Such as:

```
struct some_structure homer;
struct some_structure *homer_pointer;
homer_pointer=&homer;
```

This defines homer_pointer to point to the structure homer. Now, when you use the pointer to reference something in the structure, the record selector now becomes **`->`** instead of a period.

```
homer_pointer->an_element=5;
```

This is the same as:

```
homer.an_element=5;
```

The void pointer can represent an unknown pointer type.

```
void *joe;
```

This is a pointer to an undetermined type.

1.4.2 Typecasting

Typecasting allows a variable to act like a variable of another type. The method of typecasting is done by prefixing the variable type enclosed by parenthesis before the variable name. The actual variable is not modified.

Example:

```
float index=3; int loop=(int)index;
```

This causes index to be typecasted to act like an **int**.

Previous Section 1.3 Functions | Table of Contents | Index | Next Section 1.5 Operators

Previous Section 1.4 References | Table of Contents | Index | Next Section 1.6 Statements

1.5.1 Postfix

Postfix operators are operators that are suffixed to an expression.

operand`++;`
This causes the value of the *operand* to be returned. After the result is obtained, the value of the operand is incremented by 1.

operand`--;`
This is the same but the value of the operand is decremented by 1.

Examples:

```
int joe=3;
joe++;
```

The value of **`joe`** is now 4.

```
printf("%i",joe++);
```

This outputs the number 4. The value of **`joe`** is now 5.

1.5.2 Unary and Prefix

Prefix operators are operators that are prefixed to an expression.

`++`*operand*`;`
This causes the value of the operand to be incremented by 1. Its new value is then returned.

`--`*operand*`;`
This is the same but the value of the operand is decremented by 1.

`!`*operand*

Returns the logical NOT operation on the operand. A true operand returns false, a false operand returns true. Also known as the bang operand.

~*operand*
Returns the compliment of the operand. The returned value is the operand with its bits reversed (1's become 0's, 0's become 1's).

Examples:

```
int bart=7;
printf("%i",--bart);
```
This outputs the number 6. The value of bart is now 6.

```
int lisa=1;
printf("%i",!lisa);
```
This outputs 0 (false).

1.5.3 Normal

There are several normal operators which return the result defined for each:

expression1 **+** *expression*
The result of this is the sum of the two expressions.

expression1 **-** *expression2*
The result of this is the value of *expression2* subtracted from *expression1*.

expression1 ***** *expression2*
The result of this is the value of *expression1* multiplied by *expression2*.

expression1 **/** *expression2*
The result of this is the value of *expression1* divided by *expression2*.

expression1 **%** *expression2*
The result of this is the value of the remainder after dividing *expression1* by *expression2*. Also called the modulo operator.

expression1 **&** *expression2*
Returns a bitwise AND operation done on *expression1* and *expression2*. The result is a value the same size as the expressions with its bits modified using the following rules: Both bits must be 1 (on) to result in 1 (on), otherwise the result is 0 (off).

e1	e2	Result

0	0	0
0	1	0
1	0	0
1	1	1

expression1 **|** *expression2*
Returns a bitwise OR operation done on *expression1* and *expression2*. The result is a value the same size as the expressions with its bits modified using the following rules: Both bits must be 0 (off) to result in 0 (off), otherwise the result is 1 (on).

e1	e2	Result
0	0	0
0	1	1
1	0	1
1	1	1

expression1 **^** *expression2*
Returns a bitwise XOR operation done on *expression1* and *expression2*. The result is a value the same size as the expressions with its bits modified using the following rules: If both bits are the same, then the result is 0 (off), otherwise the result is 1 (on).

e1	e2	Result
0	0	0
0	1	1
1	0	1
1	1	0

expression1 **>>** *shift_value*
Returns *expression1* with its bits shifted to the right by the *shift_value*. The leftmost bits are replaced with zeros if the value is nonnegative or unsigned. This result is the integer part of *expression1* divided by 2 raised to the power of *shift_value*. If *expression1* is signed, then the result is implementation specific.

expression1 **<<** *shift_value*
Returns *expression1* with its bits shifted to the left by the *shift_value*. The rightmost bits are replaced with zeros. This result is the value of *expression1* multiplied by the value of 2 raised to the power of *shift_value*. If *expression1* is signed, then the result is implementation specific.

1.5.4 Boolean

The boolean operators return either 1 (true) or 0 (false).

expression1 **&&** *expression2*
Returns the logical AND operation of *expression1* and *expression2*. The result is 1 (true) if both expressions are true, otherwise the result is 0 (false).

e1	e2	Result
0	0	0
0	1	0
1	0	0
1	1	1

expression1 **||** *expression2*
Returns the logical OR operation of *expression1* and *expression2*. The result is 0 (false) if bother expressions are false, otherwise the result is 1 (true).

e1	e2	Result
0	0	0
0	1	1
1	0	1
1	1	1

expression1 **<** *expression2*
Returns 1 (true) if *expression1* is less than *expression2*, otherwise the result is 0 (false).

expression1 **>** *expression2*
Returns 1 (true) if *expression1* is greater than *expression2*, otherwise the result is 0 (false).

expression1 **<=** *expression2*
Returns 1 (true) if *expression1* is less than or equal to *expression2*, otherwise the result is 0 (false).

expression1 **>=** *expression2*
Returns 1 (true) if *expression1* is greater than or equal to *expression2*, otherwise the result is 0 (false).

expression1 **==** *expression2*
Returns 1 (true) if *expression1* is equal to *expression2*, otherwise the result is 0 (false).

expression1 **!=** *expression2*
Returns 1 (true) if *expression*1 is not equal to *expression2*, otherwise the result is 0 (false).

1.5.5 Assignment

An assignment operator stores the value of the right expression into the left expression.

expression1 **=** *expression2*
The value of *expression2* is stored in *expression1*.

expression1 ***=** *expression2*
The value of *expression1* times *expression2* is stored in *expression1*.

expression1 **/=** *expression2*
The value of *expression1* divided by *expression2* is stored in *expression1*.

expression1 **%=** *expression2*
The value of the remainder of *expression1* divided by *expression2* is stored in *expression1*.

expression1 **+=** *expression2*
The value of *expression1* plus *expression2* is stored in *expression1*.

expression1 **-=** *expression2*
The value of *expression1* minus *expression2* is stored in *expression1*.

expression1 **<<=** *shift_value*
The value of *expression1*'s bits are shifted to the left by *shift_value* and stored in *expression1*.

expression1 **>>=** *shift_value*
The value of *expression1*'s bits are shifted to the right by *shift_value* and stored in *expression1*.

expression1 **&=** *expression2*
The value of the bitwise AND of *expression1* and *expression2* is stored in *expression1*.

e1	e2	Result
0	0	0
0	1	0
1	0	0
1	1	1

expression1 **^=** *expression2*
The value of the bitwise XOR of *expression1* and *expression2* is stored in *expression1*.

e1	e2	Result
0	0	0
0	1	1
1	0	1
1	1	0

expression1 **|=** *expression2*
The value of the bitwise OR of *expression1* and *expression2* is stored in *expression1*.

e1	e2	Result
0	0	0
0	1	1
1	0	1
1	1	1

1.5.6 Precedence

The operators have a set order of precedence during evaluation. Items encapsulated in parenthesis are evaluated first and have the highest precedence. The following chart shows the order of precedence with the items at the top having highest precedence.

Operator	Name
!	Logical NOT. Bang.
++ --	Increment and decrement operators.
* / %	Multiplicative operators.
+ -	Additive operators.
<< >>	Shift operators.
< > <= >=	Inequality comparators.
== !=	Equality comparators
&	Bitwise AND.
^	Bitwise XOR.
|	Bitwise OR.
&&	Logical AND.

||	Logical OR.
?:	Conditional.
= *op*=	Assignment.

Examples:

```
17 * 5 + !(1+1) && 0
```

Evaluates to 0 (false).

```
5+7<4
```

Evaluates to 1 (true).

```
a<b<c
```

Same as (a<b)<c.

Previous Section 1.4 References | Table of Contents | Index | Next Section 1.6 Statements

Previous Section 1.5 Operators | Table of Contents | Index | Next Section 1.7 Preprocessing Directives

1.6.1 if

The if statement evaluates an expression. If that expression is true, then a statement is executed. If an else clause is given and if the expression is false, then the else's statement is executed.

Syntax:

if(*expression* **)** *statement1***;**

or

if(*expression* **)** *statement1***;**
else *statement2* **;**

Examples:

```
if(loop<3) counter++;

if(x==y)
  x++;
else
  y++;

if(z>x)
 {
  z=5;
  x=3;
 }
else
 {
  z=3;
  x=5;
 }
```

1.6.2 switch

A switch statement allows a single variable to be compared with several possible constants. If the variable matches one of the constants, then a execution jump is made to that point. A constant can not appear more than once, and there can only be one default expression.

Syntax:

```
switch ( variable )
 {
  case const:
   statements...;
  default:
   statements...;
 }
```

Examples:

```
switch(betty)
 {
  case 1:
    printf("betty=1\n");
  case 2:
    printf("betty=2\n");
    break;
  case 3:
    printf("betty=3\n");
    break;
  default:
    printf("Not sure.\n");
 }
```

If betty is 1, then two lines are printed: betty=1 and betty=2. If betty is 2, then only one line is printed: betty=2. If betty=3, then only one line is printed: betty=3. If betty does not equal 1, 2, or 3, then "Not sure." is printed.

1.6.3 while

The while statement provides an iterative loop.

Syntax:

while(*expression* **)** *statement...*

statement is executed repeatedly as long as *expression* is true. The test on *expression* takes place before each execution of *statement*.
Examples:

```
while(*pointer!='j') pointer++;

while(counter<5)
 {
  printf("counter=%i",counter);
  counter++;
 }
```

1.6.4 do

The do...while construct provides an iterative loop.

Syntax:

do *statement...* **while(** *expression* **);**

statement is executed repeatedly as long as *expression* is true. The test on *expression* takes place after each execution of *statement*.

Examples:

```
do {
  betty++;
  printf("%i",betty);
} while (betty<100);
```

1.6.5 for

The for statement allows for a controlled loop.

Syntax:

for(*expression1* ; *expression2* ; *expression3* **)** *statement...*

expression1 is evaluated before the first iteration. After each iteration, *expression3* is evaluated. Both *expression1* and *expression3* may be ommited. If *expression2* is ommited, it is assumed to be 1. *statement* is executed repeatedly until the value of *expression2* is 0. The test on *expression2* occurs before each execution of *statement*.

Examples:

```
for(loop=0;loop<1000;loop++)
  printf("%i\n",loop);
```

Prints numbers 0 through 999.

```
for(x=3, y=5; x<100+y; x++, y--)
 {
  printf("%i\n",x);
  some_function();
 }
```

Prints numbers 3 through 53. some_function is called 51 times.

1.6.6 goto

The goto statement transfers program execution to some label within the program.

Syntax:

goto *label***;**
....
*label***:**

Examples:

```
goto skip_point;
printf("This part was skipped.\n");
skip_point:
```

```
printf("Hi there!\n");
```

Only the text "Hi there!" is printed.

1.6.7 continue

The continue statement can only appear in a loop body. It causes the rest of the statement body in the loop to be skipped.

Syntax:

```
continue;
```

Examples:

```
for(loop=0;loop<100;loop++)
 {
  if(loop==50)
    continue;
  printf("%i\n",loop);
 }
```

The numbers 0 through 99 are printed except for 50.

```
joe=0;
while(joe<1000)
 {
 for(zip=0;zip<100;zip++)
  {
   if(joe==500)
    continue;
   printf("%i\n",joe);
  }
 joe++;
}
```

Each number from 0 to 999 is printed 100 times except for the number 500 which is not printed at all.

1.6.8 break

The break statement can only appear in a switch body or a loop body. It causes the execution of the current enclosing switch or loop body to terminate.

Syntax:

```
break;
```

Examples:

```
switch(henry)
 {
 case 1: print("Hi!\n");
         break;
 case 2: break;
 }
```

If henry is equal to 2, nothing happens.

```
for(loop=0;loop<50;loop++)
 {
  if(loop==10)
    break;
  printf("%i\n",loop);
 }
```

Only numbers 0 through 9 are printed.

1.6.9 return

The return statement causes the current function to terminate. It can return a value to the calling function. A return statement can not appear in a function whose return type is void. If the value returned has a type different from that of the function's return type, then the value is converted. Using the return statement without an expression creates an undefined result. Reaching the } at the end of the function is the same as returning without an expression.

Syntax:

return *expression*;

Examples:

```
int alice(int x, int y)
 {
  if(x<y)
    return(1);
  else
    return(0);
}
```

Previous Section 1.5 Operators | Table of Contents | Index | Next Section 1.7 Preprocessing Directives

Previous Section 1.6 Statements | Table of Contents | Index | Next Section 2.1 assert.h

1.7.1 #if, #elif, #else, #endif

These preprocessing directives create conditional compiling parameters that control the compiling of the source code. They must begin on a separate line.

Syntax:

```
#if constant_expression
#else
#endif

or

#if constant_expression
#elif constant_expression
#endif
```

The compiler only compiles the code after the **#if** expression if the *constant_expression* evaluates to a non-zero value (true). If the value is 0 (false), then the compiler skips the lines until the next **#else**, **#elif**, or **#endif**. If there is a matching **#else**, and the *constant_expression* evaluated to 0 (false), then the lines between the **#else** and the **#endif** are compiled. If there is a matching **#elif**, and the preceding **#if** evaluated to false, then the *constant_expression* after that is evaluated and the code between the **#elif** and the **#endif** is compiled only if this expression evaluates to a non-zero value (true).

Examples:

```
int main(void)
 {
  #if 1
    printf("Yabba Dabba Do!\n");
  #else
    printf("Zip-Bang!\n");
  #endif
  return 0;
}
```

Only "Yabba Dabba Do!" is printed.

```
int main(void)
 {
  #if 1
    printf("Checkpoint1\n");
  #elif 1
    printf("Checkpoint2\n");
  #endif
  return 0;
}
```

Only "Checkpoint1" is printed. Note that if the first line is #if 0, then only "Checkpoint2" would be printed.

```
#if OS==1
  printf("Version 1.0");
#elif OS==2
  printf("Version 2.0");
#else
  printf("Version unknown");
#endif
```

Prints according to the setting of OS which is defined with a #define.

1.7.2 #define, #undef, #ifdef, #ifndef

The preprocessing directives **`#define`** and **`#undef`** allow the definition of identifiers which hold a certain value. These identifiers can simply be constants or a macro function. The directives **`#ifdef`** and **`#ifndef`** allow conditional compiling of certain lines of code based on whether or not an identifier has been defined.

Syntax:

`#define` *identifier replacement-code*

`#undef` *identifier*

`#ifdef` *identifier*
`#else` or **`#elif`**
`#endif`

#ifndef *identifier*
#else or **#elif**
#endif

#ifdef *identifier* is the same is **#if defined(** *identifier***)**.
#ifndef *identifier* is the same as **#if !defined(***identifier***)**.
An identifier defined with **#define** is available anywhere in the source code until a **#undef** is reached.
A function macro can be defined with **#define** in the following manner:

#define *identifier***(***parameter-list***) (***replacement-text***)**

The values in the *parameter-list* are replaced in the *replacement-text*.

Examples:

```
#define PI 3.141
printf("%f",PI);

#define DEBUG
#ifdef DEBUG
  printf("This is a debug message.");
#endif

#define QUICK(x) printf("%s\n",x);
QUICK("Hi!")

#define ADD(x, y) x + y
z=3 * ADD(5,6)
```

This evaluates to 21 due to the fact that multiplication takes precedence over addition.

```
#define ADD(x,y) (x + y)
z=3 * ADD(5,6)
```

This evaluates to 33 due to the fact that the summation is encapsulated in parenthesis which takes precedence over multiplication.

1.7.3 #include

The **#include** directive allows external header files to be processed by the compiler.

Syntax:

#include <*header-file***>**

or

#include "*source-file***"**

When enclosing the file with < and >, then the implementation searches the known header directories for the file (which is implementation-defined) and processes it. When enclosed with double quotation marks, then the entire contents of the source-file is replaced at this point. The searching manner for the file is implementation-specific.

Examples:

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

1.7.4 #line

The **#line** directive allows the current line number and the apparent name of the current sourcecode filename to be changed.

Syntax:

#line *line-number filename*

Note that if the filename is not given, then it stays the same. The line number on the current line is one greater than the number of new-line characters (so the first line number is 1).
Examples:

```
#line 50 user.c

#line 23
```

1.7.5 #error

The **#error** directive will cause the compiler to halt compiling and return with the specified error message.

Syntax:

`#error` *message*

Examples:

```
#ifndef VERSION
#error Version number not specified.
#endif
```

1.7.6 #pragma

This **`#pragma`** directive allows a directive to be defined. Its effects are implementation-defined. If the pragma is not supported, then it is ignored.

Syntax:

`#pragma` *directive*

1.7.7 Predefined Macros

The following macros are already defined by the compiler and cannot be changed.

__LINE__	A decimal constant representing the current line number.
__FILE__	A string representing the current name of the source code file.
__DATE__	A string representing the current date when compiling began for the current source file. It is in the format "mmm dd yyyy", the same as what is generated by the asctime function.
__TIME__	A string literal representing the current time when cimpiling began for the current source file. It is in the format "hh:mm:ss", the same as what is generated by the asctime function.
__STDC__	The decimal constant 1. Used to indicate if this is a standard C compiler.

Previous Section 1.6 Statements | Table of Contents | Index | Next Section 2.1 assert.h

Previous Section 1.7 Preprocessing Directives | Table of Contents | Index | Next Section 2.2 ctype.h

2.1 assert.h

The assert header is used for debugging purposes.

Macros:

```
assert();
```

External References:

```
NDEBUG
```

2.1.1 assert

Declaration:

`void assert(int` *expression*`);`

The assert macro allows diagnostic information to be written to the standard error file.

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then calls the abort function. If the identifier **`NDEBUG`** ("no debug") is defined with **`#define NDEBUG`** then the macro assert does nothing.

Common error outputting is in the form:

`Assertion failed:` *expression*`,` **`file`** *filename*`,` **`line`** *line-number*

Example:

```
#include<assert.h>

void open_record(char *record_name)
```

```
{
  assert(record_name!=NULL);
  /* Rest of code */
}

int main(void)
{
  open_record(NULL);
}
```

Previous Section 1.7 Preprocessing Directives | Table of Contents | Index | Next Section 2.2 ctype.h

Previous Section 2.1 assert.h | Table of Contents | Index | Next Section 2.3 errno.h

2.2 ctype.h

The ctype header is used for testing and converting characters. A control character refers to a character that is not part of the normal printing set. In the ASCII character set, the control characters are the characters from 0 (NUL) through 0x1F (US), and the character 0x7F (DEL). Printable characters are those from 0x20 (space) to 0x7E (tilde).

Functions:

```
isalnum();
isalpha();
iscntrl();
isdigit();
isgraph();
islower();
isprint();
ispunct();
isspace();
isupper();
isxdigit();
tolower();
toupper();
```

2.2.1 is... Functions

Declarations:

```
int isalnum(int character);
int isalpha(int character);
int iscntrl(int character);
int isdigit(int character);
int isgraph(int character);
int islower(int character);
int isprint(int character);
int ispunct(int character);
```

```
int isspace(int character);
int isupper(int character);
int isxdigit(int character);
```

The is... functions test the given character and return a nonzero (true) result if it satisfies the following conditions. If not, then 0 (false) is returned.

Conditions:

isalnum	a letter (A to Z or a to z) or a digit (0 to 9)
isalpha	a letter (A to Z or a to z)
iscntrl	any control character (0x00 to 0x1F or 0x7F)
isdigit	a digit (0 to 9)
isgraph	any printing character except for the space character (0x21 to 0x7E)
islower	a lowercase letter (a to z)
isprint	any printing character (0x20 to 0x7E)
ispunct	any punctuation character (any printing character except for space character or isalnum)
isspace	a whitespace character (space, tab, carriage return, new line, vertical tab, or formfeed)
isupper	an uppercase letter (A to Z)
isxdigit	a hexadecimal digit (0 to 9, A to F, or a to f)

2.2.2 to... Functions

Declarations:

```
int tolower(int character);
int toupper(int character);
```

The to... functions provide a means to convert a single character. If the character matches the appropriate condition, then it is converted. Otherwise the character is returned unchanged.

Conditions:

tolower	If the character is an uppercase character (A to Z), then it is converted to lowercase (a to z)
toupper	If the character is a lowercase character (a to z), then it is converted to uppercase (A to Z)

Example:

```
#include<ctype.h>
```

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

int main(void)
{
  int loop;
  char string[]="THIS IS A TEST";

  for(loop=0;loop<strlen(string);loop++)
    string[loop]=tolower(string[loop]);

  printf("%s\n",string);
  return 0;
}
```

Previous Section 2.1 assert.h | Table of Contents | Index | Next Section 2.3 errno.h

Previous Section 2.2 ctype.h | Table of Contents | Index | Next Section 2.4 float.h

2.3 errno.h

The errno header is used as a general error handler.

Macros:

EDOM
ERANGE

Variables:

errno

2.3.1 EDOM

Declaration:

#define EDOM *some_value*

EDOM is an identifier macro declared with **#define**. Its value represents a domain error which is returned by some math functions when a domain error occurs.

2.3.2 ERANGE

Declaration:

#define ERANGE *some_value*

ERANGE is an identifier macro declared with **#define**. Its value represents a range error which is returned by some math functions when a range error occurs.

2.3.3 errno

Declaration:

```
int errno;
```

The **errno** variable has a value of zero at the beginning of the program. If an error occurs, then this variable is given the value of the error number.

Previous Section 2.2 ctype.h | Table of Contents | Index | Next Section 2.4 float.h

Previous Section 2.3 errno.h | Table of Contents | Index | Next Section 2.5 limits.h

2.4 float.h

The float header defines the minimum and maximum limits of floating-point number values.

2.4.1 Defined Values

A floating-point number is defined in the following manner:

sign value E *exponent*

Where *sign* is plus or minus, *value* is the value of the number, and *exponent* is the value of the exponent.

The following values are defined with the **`#define`** directive. These values are implementation-specific, but may not be any lower than what is given here. Note that in all instances **`FLT`** refers to type float, **`DBL`** refers to double, and **`LDBL`** refers to long double.

`FLT_ROUNDS`	Defines the way floating-point numbers are rounded. **`-1`** indeterminable **`0`** toward zero **`1`** to nearest **`2`** toward positive infinity **`3`** toward negative infinity
`FLT_RADIX 2`	Defines the base (radix) representation of the exponent (i.e. base-2 is binary, base-10 is the normal decimal representation, base-16 is Hex).
`FLT_MANT_DIG` **`DBL_MANT_DIG`** **`LDBL_MANT_DIG`**	Defines the number of digits in the number (in the **`FLT_RADIX`** base).
`FLT_DIG 6` **`DBL_DIG 10`** **`LDBL_DIG 10`**	The maximum number decimal digits (base-10) that can be represented without change after rounding.

FLT_MIN_EXP **DBL_MIN_EXP** **LDBL_MIN_EXP**	The minimum negative integer value for an exponent in base FLT_RADIX.
FLT_MIN_10_EXP -37 **DBL_MIN_10_EXP -37** **LDBL_MIN_10_EXP -37**	The minimum negative integer value for an exponent in base 10.
FLT_MAX_EXP **DBL_MAX_EXP** **LDBL_MAX_EXP**	The maximum integer value for an exponent in base FLT_RADIX.
FLT_MAX_10_EXP +37 **DBL_MAX_10_EXP +37** **LDBL_MAX_10_EXP +37**	The maximum integer value for an exponent in base 10.
FLT_MAX 1E+37 **DBL_MAX 1E+37** **LDBL_MAX 1E+37**	Maximum finite floating-point value.
FLT_EPSILON 1E-5 **DBL_EPSILON 1E-9** **LDBL_EPSILON 1E-9**	Least significant digit representable.
FLT_MIN 1E-37 **DBL_MIN 1E-37** **LDBL_MIN 1E-37**	Minimum floating-point value.

Previous Section 2.3 errno.h | Table of Contents | Index | Next Section 2.5 limits.h

Previous Section 2.4 float.h | Table of Contents | Index | Next Section 2.6 locale.h

2.5 limits.h

The limits header defines the characteristics of variable types.

2.5.1 Defined Values

The following values are defined with the **#define** directive. These values are implementation-specific, but may not be any lower than what is given here.

CHAR_BIT 8	Number of bits in a byte.
SCHAR_MIN -127	Minimum value for a signed char.
SCHAR_MAX +127	Maximum value for a signed char.
UCHAR_MAX 255	Maximum value for an unsigned char.
CHAR_MIN **CHAR_MAX**	These define the minimum and maximum values for a char. If a char is being represented as a signed integer, then their values are the same as the signed char (**SCHAR**) values. Otherwise **CHAR_MIN** is 0 and **CHAR_MAX** is the same as **UCHAR_MAX**.
MB_LEN_MAX 1	Maximum number of bytes in a multibyte character.
SHRT_MIN -32767	Minimum value for a short int.
SHRT_MAX +32767	Maximum value for a short int.
USHRT_MAX 65535	Maximum value for an unsigned short int.
INT_MIN -32767	Minimum value for an int.
INT_MAX +32767	Maximum value for an int.
UINT_MAX 65535	Maximum value for an unsigned int.
LONG_MIN -2147483647	Minimum value for a long int.
LONG_MAX +2147483647	Maximum value for a long int.
ULONG_MAX 4294967295	Maximum value for an unsigned long int.

Previous Section 2.4 float.h | Table of Contents | Index | Next Section 2.6 locale.h

Previous Section 2.5 limits.h | Table of Contents | Index | Next Section 2.7 math.h

2.6 locale.h

The locale header is useful for setting location specific information.

Variables:

```
struct lconv
```

Macros:

```
NULL
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

Functions:

```
localeconv();
setlocale();
```

2.6.1 Variables and Definitions

The **lconv** structure contains the following variables in any order. The use of this structure is described in **2.6.3 localeconv**.

```
char *decimal_point;
char *thousands_sep;
char *grouping;
char *int_curr_symbol;
char *currency_symbol;
char *mon_decimal_point;
char *mon_thousands_sep;
```

```
char *mon_grouping;
char *positive_sign;
char *negative_sign;
char int_frac_digits;
char frac_digits;
char p_cs_precedes;
char p_sep_by_space;
char n_cs_precedes;
char n_sep_by_space;
char p_sign_posn;
char n_sign_posn;
```

The **LC_** macros are described in **2.6.2 setlocale**. **NULL** is the value of a null pointer constant.

2.6.2 setlocale

Declaration:

char *setlocale(int *category*, **const char *** *locale*);

Sets or reads location dependent information.

category can be one of the following:

LC_ALL	Set everything.
LC_COLLATE	Affects strcoll and strxfrm functions.
LC_CTYPE	Affects all character functions.
LC_MONETARY	Affects the monetary information provided by localeconv function.
LC_NUMERIC	Affects decimal-point formatting and the information provided by **localeconv** function.
LC_TIME	Affects the strftime function.

A value of "C" for locale sets the locale to the normal C translation environment settings (default). A null value ("") sets the native environment settings. A null pointer (NULL) causes setlocale to return a pointer to the string associated with this category for the current settings (no changes occur). All other values are implementation-specific.

After a successful set, **setlocale** returns a pointer to a string which represents the previous location setting. On failure it returns NULL.

Example:

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

int main(void)
{
  char *old_locale;

  old_locale=setlocale(LC_ALL,"C");
  printf("The preivous setting was %s.\n",old_locale);
  return 0;
}
```

2.6.3 localeconv

Declaration:

```
struct lconv *localeconv(void);
```

Sets the structure **lconv** to represent the current location settings.

The string pointers in the structure may point to a null string ("") which indicates that the value is not available. The char types are nonnegative numbers. If the value is **CHAR_MAX**, then the value is not available.

lconv variables:

char *decimal_point	Decimal point character used for non-monetary values.
char *thousands_sep	Thousands place separator character used for non-monetary values.
char *grouping	A string that indicates the size of each group of digits in non-monetary quantities. Each character represents an integer value which designates the number of digits in the current group. A value of 0 means that the previous value is to be used for the rest of the groups.
char *int_curr_symbol	A string of the international currency symbols used. The first three characters are those specified by ISO 4217:1987 and the fourth is the character which separates the currency symbol from the monetary quantity.
char *currency_symbol	The local symbol used for currency.

char *mon_decimal_point	The decimal point character used for monetary values.
char *mon_thousands_sep	The thousands place grouping character used for monetary values.
char *mon_grouping	A string whose elements define the size of the grouping of digits in monetary values. Each character represents an integer value which designates the number of digits in the current group. A value of 0 means that the previous value is to be used for the rest of the groups.
char *positive_sign	The character used for positive monetary values.
char *negative_sign	The character used for negative monetary values.
char int_frac_digits	Number of digits to show after the decimal point in international monetary values.
char frac_digits	Number of digits to show after the decimal point in monetary values.
char p_cs_precedes	If equal to 1, then the **currency_symbol** appears before a positive monetary value. If equal to 0, then the **currency_symbol** appears after a positive monetary value.
char p_sep_by_space	If equal to 1, then the **currency_symbol** is separated by a space from a positive monetary value. If equal to 0, then there is no space between the **currency_symbol** and a positive monetary value.
char n_cs_precedes	If equal to 1, then the **currency_symbol** precedes a negative monetary value. If equal to 0, then the **currency_symbol succeeds a negative monetary value.**
char n_sep_by_space	If equal to 1, then the **currency_symbol** is separated by a space from a negative monetary value. If equal to 0, then there is no space between the **currency_symbol** and a negative monetary value.
char p_sign_posn	Represents the position of the **positive_sign** in a positive monetary value.
char n_sign_posn	Represents the position of the **negative_sign** in a negative monetary value.

The following values are used for **p_sign_posn** and **n_sign_posn**:

0 Parentheses encapsulate the value and the currency_symbol.
1 The sign precedes the value and currency_symbol.
2 The sign succeeds the value and currency_symbol.
3 The sign immediately precedes the value and currency_symbol.
4 The sign immediately succeeds the value and currency_symbol.

Example:

```
#include<locale.h>
#include<stdio.h>
int main(void)
{
  struct lconv locale_structure;
  struct lconv *locale_ptr=&locale_structure;

  locale_ptr=lcoaleconv();
  printf("Decimal point:                  %s",locale_ptr-
>decimal_point);
  printf("Thousands Separator:            %s",locale_ptr-
>thousands_sep);
  printf("Grouping:                       %s",locale_ptr-
>grouping);
  printf("International Currency Symbol:  %s",locale_ptr-
>int_curr_symbol);
  printf("Currency Symbol:                %s",locale_ptr-
>currency_symbol);
  printf("Monetary Decimal Point:         %s",locale_ptr-
>mon_decimal_point);
  printf("Monetary Thousands Separator:   %s",locale_ptr-
>mon_thousands_sep);
  printf("Monetary Grouping:              %s",locale_ptr-
>mon_grouping);
  printf("Monetary Positive Sign:         %s",locale_ptr-
>positive_sign);
  printf("Monetary Negative Sign:         %s",locale_ptr-
>negative_sign);
  printf("Monetary Intl Decimal Digits:   %c",locale_ptr-
>int_frac_digits);
  printf("Monetary Decimal Digits:        %c",locale_ptr-
>frac_digits);
  printf("Monetary + Precedes:            %c",locale_ptr-
>p_cs_precedes);
  printf("Monetary + Space:               %c",locale_ptr-
>p_sep_by_space);
  printf("Monetary - Precedes:            %c",locale_ptr-
>n_cs_precedes);
  printf("Monetary - Space:               %c",locale_ptr-
>n_sep_by_space);
  printf("Monetary + Sign Posn:           %c",locale_ptr-
```

```
>p_sign_posn);
  printf("Monetary - Sign Posn:            %c",locale_ptr-
>n_sign_posn);
}
```

Previous Section 2.5 limits.h | Table of Contents | Index | Next Section 2.7 math.h

Previous Section 2.6 locale.h | Table of Contents | Index | Next Section 2.8 setjmp.h

2.7 math.h

The math header defines several mathematic functions.

Macros:

```
HUGE_VAL
```

Functions:

```
acos();
asin();
atan();
atan2();
ceil();
cos();
cosh();
exp();
fabs();
floor();
fmod();
frexp();
ldexp();
log();
log10();
modf();
pow();
sin();
sinh();
sqrt();
tan();
tanh();
```

2.7.1 Error Conditions

All math.h functions handle errors similarly.

In the case that the argument passed to the function exceeds the range of that function, then the variable **errno** is set to **EDOM**. The value that the function returns is implementation specific.

In the case that the value being returned is too large to be represented in a double, then the function returns the macro **HUGE_VAL**, and sets the variable **errno** to **ERANGE** to represent an overflow. If the value is too small to be represented in a double, then the function returns zero. In this case whether or not **errno** is set to **ERANGE** is implementation specific.

errno, **EDOM**, and **ERANGE** are defined in the errno.h header.

Note that in all cases when it is stated that there is no range limit, it is implied that the value is limited by the minimum and maximum values of type double.

2.7.2 Trigonometric Functions

2.7.2.1 acos

Declaration:

> **double acos(double** *x***);**
>
> Returns the arc cosine of *x* in radians.

Range:

> The value *x* must be within the range of -1 to +1 (inclusive). The returned value is in the range of 0 to pi (inclusive).

2.7.2.2 asin

Declaration:

> **double asin(double** *x***);**
>
> Returns the arc sine of *x* in radians.

Range:

The value of *x* must be within the range of -1 to +1 (inclusive). The returned value is in the range of -p/2 to +p/2 (inclusive).

2.7.2.3 atan

Declaration:

double atan(double *x***);**

Returns the arc tangent of *x* in radians.

Range:

The value of *x* has no range. The returned value is in the range of -p/2 to +p/2 (inclusive).

2.7.2.4 atan2

Declaration:

double atan2(doubly *y***, double** *x***);**

Returns the arc tangent in radians of *y/x* based on the signs of both values to determine the correct quadrant.

Range:

Both *y* and *x* cannot be zero. The returned value is in the range of -p/2 to +p/2 (inclusive).

2.7.2.5 cos

Declaration:

double cos(double *x***);**

Returns the cosine of a radian angle *x*.

Range:

The value of *x* has no range. The returned value is in the range of -1 to +1 (inclusive).

2.7.2.6 cosh

Declaration:

double cosh(double *x*);

Returns the hyperbolic cosine of *x*.

Range:

There is no range limit on the argument or return value.

2.7.2.7 sin

Declaration:

double sin(double *x*);

Returns the sine of a radian angle *x*.

Range:

The value of *x* has no range. The returned value is in the range of -1 to +1 (inclusive).

2.7.2.8 sinh

Declaration:

double sinh(double *x*);

Returns the hyperbolic sine of *x*.

Range:

There is no range limit on the argument or return value.

2.7.2.9 tan

Declaration:

double tan(double *x*);

Returns the tangent of a radian angle *x*.

Range:

There is no range limit on the argument or return value.

2.7.2.10 tanh

Declaration:

double tanh(double *x*);

Returns the hyperbolic tangent of *x*.

Range:

The value of *x* has no range. The returned value is in the range of -1 to +1 (inclusive).

2.7.3 Exponential, Logarithmic, and Power Functions

2.7.3.1 exp

Declaration:

double exp(double *x*);

Returns the value of e raised to the *x*th power.

Range:

There is no range limit on the argument or return value.

2.7.3.2 frexp

Declaration:

double frexp(double *x*, **int ***_exponent_**);**

The floating-point number *x* is broken up into a mantissa and exponent.
The returned value is the mantissa and the integer pointed to by *exponent* is the exponent.
The resultant value is `x=mantissa * 2^exponent`.

Range:

The mantissa is in the range of .5 (inclusive) to 1 (exclusive).

2.7.3.3 ldexp

Declaration:

double ldexp(double *x*, **int** *exponent*);

Returns *x* multiplied by 2 raised to the power of *exponent*.
`x*2^exponent`

Range:

There is no range limit on the argument or return value.

2.7.3.4 log

Declaration:

double log(double *x*);

Returns the natural logarithm (base-e logarithm) of *x*.

Range:

There is no range limit on the argument or return value.

2.7.3.5 log10

Declaration:

double log10(double *x*);

Returns the common logarithm (base-10 logarithm) of *x*.

Range:

There is no range limit on the argument or return value.

2.7.3.6 modf

Declaration:

double modf(double *x*, **double *** *integer*);

Breaks the floating-point number *x* into integer and fraction components.
The returned value is the fraction component (part after the decimal), and sets *integer* to the integer component.

Range:

There is no range limit on the argument or return value.

2.7.3.7 pow

Declaration:

double pow(double *x*, **double** *y*);

Returns *x* raised to the power of *y*.

Range:

x cannot be negative if *y* is a fractional value. *x* cannot be zero if *y* is less than or equal to zero.

2.7.3.8 sqrt

Declaration:

double sqrt(double *x*);

Returns the square root of *x*.

Range:

The argument cannot be negative. The returned value is always positive.

2.7.4 Other Math Functions

2.7.4.1 ceil

Declaration:

double ceil(double *x*);

Returns the smallest integer value greater than or equal to *x*.

Range:

There is no range limit on the argument or return value.

2.7.4.2 fabs

Declaration:

double fabs(double *x*);

Returns the absolute value of *x* (a negative value becomes positive, positive value is unchanged).

Range:

There is no range limit on the argument. The return value is always positive.

2.7.4.3 floor

Declaration:

double floor(double *x*);

Returns the largest integer value less than or equal to *x*.

Range:

There is no range limit on the argument or return value.

2.7.4.4 fmod

Declaration:

double fmod(double *x*, **double** *y*);

Returns the remainder of *x* divided by *y*.

Range:

There is no range limit on the return value. If *y* is zero, then either a range error will occur or the function will return zero (implementation-defined).

Previous Section 2.6 locale.h | Table of Contents | Index | Next Section 2.8 setjmp.h

Previous Section 2.7 math.h | Table of Contents | Index | Next Section 2.9 signal.h

2.8 setjmp.h

The setjmp header is used for controlling low-level calls and returns to and from functions.

Macros:

```
setjmp();
```

Functions:

```
longjmp();
```

Variables:

```
typedef jmp_buf
```

2.8.1 Variables and Definitions

The variable type **`jmp_buf`** is an array type used for holding information for **`setjmp`** and **`longjmp`**.

2.8.2 setjmp

Declaration:

`int setjmp(jmp_buf` *environment*`);`

Saves the environment into the variable *environment*. If a non-zero value is returned, then this indicates that the point in the sourcecode was reached by a **`longjmp`**. Otherwise zero is returned indicating the environment has been saved.

2.8.3 longjmp

Declaration:

void longjmp(jmp_buf *environment*, **int** *value*);

Causes the environment to be restored from a **setjmp** call where the environment variable had been saved. It causes execution to goto the **setjmp** location as if **setjmp** had returned the value of the variable *value*. The variable *value* cannot be zero. However, if zero is passed, then 1 is replaced. If the function where **setjmp** was called has terminated, then the results are undefined.

Example:

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

void some_function(jmp_buf);

int main(void)
{
  int value;
  jmp_buf environment_buffer;

  value=setjmp(environment_buffer);
  if(value!=0)
   {
    printf("Reached this point from a longjmp with value=%d.
\n",value);
    exit(0);
   }
  printf("Calling function.\n");
  some_function(environment_buffer);
  return 0;
}

void some_function(jmp_buf env_buf)
{
  longjmp(env_buf,5);
}
```

The output from this program should be:

```
Calling function.
Reached this point from a longjmp with value=5.
```

Previous Section 2.7 math.h | Table of Contents | Index | Next Section 2.9 signal.h

Previous Section 2.8 setjmp.h | Table of Contents | Index | Next Section 2.10 stdarg.h

2.9 signal.h

The signal header provides a means to handle signals reported during a program's execution.

Macros:

```
SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT
SIGFPE
SIGILL
SIGINT
SIGSEGV
SIGTERM
```

Functions:

```
signal();
raise();
```

Variables:

```
typedef sig_atomic_t
```

2.9.1 Variables and Definitions

The **sig_atomic_t** type is of type **int** and is used as a variable in a signal handler. The **SIG_** macros are used with the signal function to define signal functions.

SIG_DFL Default handler.

SIG_ERR Represents a signal error.

SIG_IGN Signal ignore.

The **SIG** macros are used to represent a signal number in the following conditions:

SIGABRT	Abnormal termination (generated by the abort function).
SIGFPE	Floating-point error (error caused by division by zero, invalid operation, etc.).
SIGILL	Illegal operation (instruction).
SIGINT	Interactive attention signal (such as ctrl-C).
SIGSEGV	Invalid access to storage (segment violation, memory violation).
SIGTERM	Termination request.

2.9.2 signal

Declaration:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Controls how a signal is handled. *sig* represents the signal number compatible with the **SIG** macros. *func* is the function to be called when the signal occurs. If func is **SIG_DFL**, then the default handler is called. If *func* is **SIG_IGN**, then the signal is ignored. If *func* points to a function, then when a signal is detected the default function is called (**SIG_DFL**), then the function is called. The function must take one argument of type **int** which represents the signal number. The function may terminate with **return**, **abort**, **exit**, or **longjmp**. When the function terminates execution resumes where it was interrupted (unless it was a **SIGFPE** signal in which case the result is undefined).

If the call to signal is successful, then it returns a pointer to the previous signal handler for the specified signal type. If the call fails, then **SIG_ERR** is returned and **errno** is set appropriately.

2.9.3 raise

Declaration

```
int raise(int sig);
```

Causes signal *sig* to be generated. The *sig* argument is compatible with the **SIG** macros.

If the call is successful, zero is returned. Otherwise a nonzero value is returned.

Example:

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

void catch_function(int);

int main(void)
{
  if(signal(SIGINT, catch_function)==SIG_ERR)
   {
    printf("An error occured while setting a signal handler.
\n");
    exit(0);
   }

  printf("Raising the interactive attention signal.\n");
  if(raise(SIGINT)!=0)
   {
    printf("Error raising the signal.\n");
    exit(0);
   }
  printf("Exiting.\n");
  return 0;
}

void catch_function(int signal)
{
  printf("Interactive attention signal caught.\n");
}
```

The output from the program should be (assuming no errors):

```
Raising the interactive attention signal.
Interactive attention signal caught.
Exiting.
```

Previous Section 2.8 setjmp.h | Table of Contents | Index | Next Section 2.10 stdarg.h

Previous Section 2.9 signal.h | Table of Contents | Index | Next Section 2.11 stddef.h

2.10 stdarg.h

The stdarg header defines several macros used to get the arguments in a function when the number of arguments is not known.

Macros:

```
va_start();
va_arg();
va_end();
```

Variables:

```
typedef va_list
```

2.10.1 Variables and Definitions

The **va_list** type is a type suitable for use in accessing the arguments of a function with the stdarg macros.

A function of variable arguments is defined with the ellipsis **(,...)** at the end of the parameter list.

2.10.2 va_start

Declaration:

```
void va_start(va_list ap, last_arg);
```

Initializes *ap* for use with the **va_arg** and **va_end** macros. *last_arg* is the last known fixed argument being passed to the function (the argument before the ellipsis).

Note that **va_start** must be called before using **va_arg** and **va_end**.

2.10.3 va_arg

Declaration:

type **va_arg(va_list** *ap*, *type*);

Expands to the next argument in the paramater list of the function with type *type*. Note that *ap* must be initialized with **va_start**. If there is no next argument, then the result is undefined.

2.10.4 va_end

Declaration:

void va_end(va_list *ap*);

Allows a function with variable arguments which used the **va_start** macro to return. If **va_end** is not called before returning from the function, the result is undefined. The variable argument list *ap* may no longer be used after a call to **va_end** without a call to **va_start**.

Example:

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

void sum(char *, int, ...);

int main(void)
{
  sum("The sum of 10+15+13 is %d.\n",3,10,15,13);
  return 0;
}

void sum(char *string, int num_args, ...)
{
  int sum=0;
  va_list ap;
  int loop;

  va_start(ap,num_args);
  for(loop=0;loop<num_args;loop++)
    sum+=va_arg(ap,int);
```

```
    printf(string,sum);
    va_end(ap);
}
```

Previous Section 2.9 signal.h | Table of Contents | Index | Next Section 2.11 stddef.h

Previous Section 2.10 stdarg.h | Table of Contents | Index | Next Section 2.12 stdio.h

2.11 stddef.h

The stddef header defines several standard definitions. Many of these definitions also appear in other headers.

Macros:

```
NULL
offsetof();
```

Variables:

```
typedef ptrdiff_t
typedef size_t
typedef wchar_t
```

2.11.1 Variables and Definitions

`ptrdiff_t` is the result of subtracting two pointers.
`size_t` is the unsigned integer result of the sizeof keyword.
`wchar_t` is an integer type of the size of a wide character constant.

`NULL` is the value of a null pointer constant.

`offsetof(`*type*`,`** *member-designator*`)`**

This results in a constant integer of type **`size_t`** which is the offset in bytes of a structure member from the beginning of the structure. The member is given by *member-designator*, and the name of the structure is given in *type*.

Example:

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

```
int main(void)
{
  struct user{
     char name[50];
     char alias[50];
     int level;
  };

  printf("level is the %d byte in the user structure.\n"),
          offsetof(struct user,level));
}
```

The output should be:

```
level is the 100 byte in the user structure.
```

Previous Section 2.10 stdarg.h | Table of Contents | Index | Next Section 2.12 stdio.h

Previous Section 2.11 stddef.h | Table of Contents | Index | Next Section 2.13 stdlib.h

2.12 stdio.h

The stdio header provides functions for performing input and output.

Macros:

```
NULL
_IOFBF
_IOLBF
_IONBF
BUFSIZ
EOF
FOPEN_MAX
FILENAME_MAX
L_tmpnam
SEEK_CUR
SEEK_END
SEEK_SET
TMP_MAX
stderr
stdin
stdout
```

Functions:

```
clearerr();
fclose();
feof();
ferror();
fflush();
fgetpos();
fopen();
fread();
freopen();
fseek();
fsetpos();
```

```
ftell();
fwrite();
remove();
rename();
rewind();
setbuf();
setvbuf();
tmpfile();
tmpnam();
fprintf();
fscanf();
printf();
scanf();
sprintf();
sscanf();
vfprintf();
vprintf();
vsprintf();
fgetc();
fgets();
fputc();
fputs();
getc();
getchar();
gets();
putc();
putchar();
puts();
ungetc();
perror();
```

Variables:

```
typedef size_t
typedef FILE
typedef fpos_t
```

2.12.1 Variables and Definitions

size_t is the unsigned integer result of the sizeof keyword.
FILE is a type suitable for storing information for a file stream.
fpos_t is a type suitable for storing any position in a file.

NULL is the value of a null pointer constant.
_IOFBF, **_IOLBF**, and **_IONBF** are used in the setvbuf function.
BUFSIZ is an integer which represents the size of the buffer used by the setbuf function.
EOF is a negative integer which indicates an end-of-file has been reached.
FOPEN_MAX is an integer which represents the maximum number of files that the system can guarantee that can be opened simultaneously.
FILENAME_MAX is an integer which represents the longest length of a char array suitable for holding the longest possible filename. If the implementation imposes no limit, then this value should be the recommended maximum value.
L_tmpnam is an integer which represents the longest length of a char array suitable for holding the longest possible temporary filename created by the tmpnam function.
SEEK_CUR, **SEEK_END**, and **SEEK_SET** are used in the fseek function.
TMP_MAX is the maximum number of unique filenames that the function tmpnam can generate.
stderr, **stdin**, and **stdout** are pointers to **FILE** types which correspond to the standard error, standard input, and standard output streams.

2.12.2 Streams and Files

Streams facilitate a way to create a level of abstraction between the program and an input/output device. This allows a common method of sending and receiving data amongst the various types of devices available. There are two types of streams: text and binary.

Text streams are composed of lines. Each line has zero or more characters and are terminated by a new-line character which is the last character in a line. Conversions may occur on text streams during input and output. Text streams consist of only printable characters, the tab character, and the new-line character. Spaces cannot appear before a newline character, although it is implementation-defined whether or not reading a text stream removes these spaces. An implementation must support lines of up to at least 254 characters including the new-line character.

Binary streams input and output data in an exactly 1:1 ratio. No conversion exists and all characters may be transferred.

When a program begins, there are already three available streams: standard input, standard output, and standard error.

Files are associated with streams and must be opened to be used. The point of I/O within a file is determined by the file position. When a file is opened, the file position points to the beginning of the file unless the file is opened for an append operation in which case the position points to the end of the file. The file position follows read and write operations to indicate where the next operation will occur.

When a file is closed, no more actions can be taken on it until it is opened again. Exiting from the main function causes all open files to be closed.

2.12.3 File Functions

2.12.3.1 clearerr

Declaration:

```
void clearerr(FILE *stream);
```

Clears the end-of-file and error indicators for the given stream. As long as the error indicator is set, all stream operations will return an error until **clearerr** or **`rewind`** is called.

2.12.3.2 fclose

Declaration:

```
int fclose(FILE *stream);
```

Closes the stream. All buffers are flushed.

If successful, it returns zero. On error it returns **`EOF`**.

2.12.3.3 feof

Declaration:

```
int feof(FILE *stream);
```

Tests the end-of-file indicator for the given stream. If the stream is at the end-of-file, then it returns a nonzero value. If it is not at the end of the file, then it returns zero.

2.12.3.4 ferror

Declaration:

```
int ferror(FILE *stream);
```

Tests the error indicator for the given stream. If the error indicator is set, then it returns a nonzero value. If the error indicator is not set, then it returns zero.

2.12.3.5 fflush

Declaration:

```
int fflush(FILE *stream);
```

Flushes the output buffer of a stream. If stream is a null pointer, then all output buffers are flushed.

If successful, it returns zero. On error it returns **EOF**.

2.12.3.6 fgetpos

Declaration:

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Gets the current file position of the stream and writes it to *pos*.

If successful, it returns zero. On error it returns a nonzero value and stores the error number in the variable **errno**.

2.12.3.7 fopen

Declaration:

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

Opens the filename pointed to by filename. The mode argument may be one of the following constant strings:

r	read text mode
w	write text mode (truncates file to zero length or creates new file)
a	append text mode for writing (opens or creates file and sets file pointer to the end-of-file)
rb	read binary mode
wb	write binary mode (truncates file to zero length or creates new file)

ab	append binary mode for writing (opens or creates file and sets file pointer to the end-of-file)
r+	read and write text mode
w+	read and write text mode (truncates file to zero length or creates new file)
a+	read and write text mode (opens or creates file and sets file pointer to the end-of-file)
r+b or **rb+**	read and write binary mode
w+b or **wb+**	read and write binary mode (truncates file to zero length or creates new file)
a+b or **ab+**	read and write binary mode (opens or creates file and sets file pointer to the end-of-file)

If the file does not exist and it is opened with read mode (**r**), then the open fails.

If the file is opened with append mode (**a**), then all write operations occur at the end of the file regardless of the current file position.

If the file is opened in the update mode (**+**), then output cannot be directly followed by input and input cannot be directly followed by output without an intervening fseek, fsetpos, rewind, or fflush.

On success a pointer to the file stream is returned. On failure a null pointer is returned.

2.12.3.8 fread

Declaration:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

Reads data from the given stream into the array pointed to by *ptr*. It reads *nmemb* number of elements of size *size*. The total number of bytes read is (**size*nmemb**).

On success the number of elements read is returned. On error or end-of-file the total number of elements successfully read (which may be zero) is returned.

2.12.3.9 freopen

Declaration:

```
FILE *freopen(const char *filename, const char *mode, FILE
*stream);
```

Associates a new filename with the given open stream. The old file in stream is closed. If an error occurs while closing the file, the error is ignored. The mode argument is the same as described in the fopen command. Normally used for reassociating stdin, stdout, or stderr.

On success the pointer to the stream is returned. On error a null pointer is returned.

2.12.3.10 fseek

Declaration:

```
int fseek(FILE *stream, long int offset, int whence);
```

Sets the file position of the stream to the given offset. The argument *offset* signifies the number of bytes to seek from the given whence position. The argument *whence* can be:

SEEK_SET Seeks from the beginning of the file.

SEEK_CUR Seeks from the current position.

SEEK_END Seeks from the end of the file.

On a text stream, whence should be **SEEK_SET** and *offset* should be either zero or a value returned from **ftell**.

The end-of-file indicator is reset. The error indicator is NOT reset.

On success zero is returned. On error a nonzero value is returned.

2.12.3.11 fsetpos

Declaration:

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Sets the file position of the given stream to the given position. The argument *pos* is a position given by the function **fgetpos**. The end-of-file indicator is cleared.

On success zero is returned. On error a nonzero value is returned and the variable **errno** is set.

2.12.3.12 ftell

Declaration:

```
long int ftell(FILE *stream);
```

Returns the current file position of the given stream. If it is a binary stream, then the value is the number of bytes from the beginning of the file. If it is a text stream, then the value is a value useable by the fseek function to return the file position to the current position.

On success the current file position is returned. On error a value of **-1L** is returned and **errno** is set.

2.12.3.13 fwrite

Declaration:

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

Writes data from the array pointed to by *ptr* to the given stream. It writes *nmemb* number of elements of size *size*. The total number of bytes written is (**size*nmemb**).

On success the number of elements writen is returned. On error the total number of elements successfully writen (which may be zero) is returned.

2.12.3.14 remove

Declaration:

```
int remove(const char *filename);
```

Deletes the given filename so that it is no longer accessible (unlinks the file). If the file is currently open, then the result is implementation-defined.

On success zero is returned. On failure a nonzero value is returned.

2.12.3.15 rename

Declaration:

```
int rename(const char *old_filename, const char *new_filename);
```

Causes the filename referred to by *old_filename* to be changed to *new_filename*. If the filename pointed

to by *new_filename* exists, the result is implementation-defined.

On success zero is returned. On error a nonzero value is returned and the file is still accessible by its old filename.

2.12.3.16 rewind

Declaration:

```
void rewind(FILE *stream);
```

Sets the file position to the beginning of the file of the given stream. The error and end-of-file indicators are reset.

2.12.3.17 setbuf

Declaration:

```
void setbuf(FILE *stream, char *buffer);
```

Defines how a stream should be buffered. This should be called after the stream has been opened but before any operation has been done on the stream. Input and output is fully buffered. The default **BUFSIZ** is the size of the buffer. The argument *buffer* points to an array to be used as the buffer. If *buffer* is a null pointer, then the stream is unbuffered.

2.12.3.18 setvbuf

Declaration:

```
int setvbuf(FILE *stream, char *buffer, int mode, size_t size);
```

Defines how a stream should be buffered. This should be called after the stream has been opened but before any operation has been done on the stream. The argument *mode* defines how the stream should be buffered as follows:

_IOFBF Input and output is fully buffered. If the buffer is empty, an input operation attempts to fill the buffer. On output the buffer will be completely filled before any information is written to the file (or the stream is closed).

_IOLBF Input and output is line buffered. If the buffer is empty, an input operation attempts to fill the buffer. On output the buffer will be flushed whenever a newline character is written.

_IONBF Input and output is not buffered. No buffering is performed.

The argument *buffer* points to an array to be used as the buffer. If *buffer* is a null pointer, then **setvbuf** uses **malloc** to create its own buffer.

The argument *size* determines the size of the array.

On success zero is returned. On error a nonzero value is returned.

2.12.3.19 tmpfile

Declaration:

```
FILE *tmpfile(void);
```

Creates a temporary file in binary update mode (wb+). The tempfile is removed when the program terminates or the stream is closed.

On success a pointer to a file stream is returned. On error a null pointer is returned.

2.12.3.20 tmpnam

Declaration:

```
char *tmpnam(char *str);
```

Generates and returns a valid temporary filename which does not exist. Up to **TMP_MAX** different filenames can be generated.

If the argument *str* is a null pointer, then the function returns a pointer to a valid filename. If the argument *str* is a valid pointer to an array, then the filename is written to the array and a pointer to the same array is returned. The filename may be up to **L_tmpnam** characters long.

2.12.4 Formatted I/O Functions

2.12.4.1 ..printf Functions

Declarations:

```
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *str, const char *format, ...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *str, const char *format, va_list arg);
```

The ..printf functions provide a means to output formatted information to a stream.

fprintf	sends formatted output to a stream
printf	sends formatted output to stdout
sprintf	sends formatted output to a string
vfprintf	sends formatted output to a stream using an argument list
vprintf	sends formatted output to stdout using an argument list
vsprintf	sends formatted output to a string using an argument list

These functions take the format string specified by the *format* argument and apply each following argument to the format specifiers in the string in a left to right fashion. Each character in the format string is copied to the stream except for conversion characters which specify a format specifier.

The string commands (**sprintf** and **vsprintf**) append a null character to the end of the string. This null character is not counted in the character count.

The argument list commands (**vfprintf**, **vprintf**, and **vsprintf**) use an argument list which is prepared by **va_start**. These commands do not call **va_end** (the caller must call it).

A conversion specifier begins with the **%** character. After the **%** character come the following in this order:

[**flags**]	Control the conversion (optional).
[**width**]	Defines the number of characters to print (optional).
[**.precision**]	Defines the amount of precision to print for a number type (optional).
[**modifier**]	Overrides the size (type) of the argument (optional).
[**type**]	The type of conversion to be applied (required).

Flags:

-	Value is left justified (default is right justified). Overrides the 0 flag.

+ Forces the sign (+ or -) to always be shown. Default is to just show the - sign. Overrides the space flag.

space Causes a positive value to display a space for the sign. Negative values still show the - sign.

Alternate form:

Conversion Character	Result
o	Precision is increased to make the first digit a zero.
X or x	Nonzero value will have 0x or 0X prefixed to it.
E, e, f, g, or G	Result will always have a decimal point.
G or g	Trailing zeros will not be removed.

0 For d, i, o, u, x, X, e, E, f, g, and G leading zeros are used to pad the field width instead of spaces. This is useful only with a width specifier. Precision overrides this flag.

Width:
The width of the field is specified here with a decimal value. If the value is not large enough to fill the width, then the rest of the field is padded with spaces (unless the 0 flag is specified). If the value overflows the width of the field, then the field is expanded to fit the value. If a ***** is used in place of the width specifer, then the next argument (which must be an **int** type) specifies the width of the field. Note: when using the ***** with the width and/or precision specifier, the width argument comes first, then the precision argument, then the value to be converted.

Precision:
The precision begins with a dot (.) to distinguish itself from the width specifier. The precision can be given as a decimal value or as an asterisk (*****). If a ***** is used, then the next argument (which is an **int** type) specifies the precision. Note: when using the ***** with the width and/or precision specifier, the width argument comes first, then the precision argument, then the value to be converted. Precision does not affect the c type.

[.precision]	Result
(*none*)	Default precision values: 1 for **d**, **i**, **o**, **u**, **x**, **X** types. The minimum number of digits to appear. 6 for **f**, **e**, **E** types. Specifies the number of digits after the decimal point. For **g** or **G** types all significant digits are shown. For **s** type all characters in string are print up to but not including the null character.
. *or* **.0**	For **d**, **i**, **o**, **u**, **x**, **X** types the default precis ion value is used unless the value is zero in which case no characters are printed. For **f**, **e**, **E** types no decimal point character or digits are printed. For **g** or **G** types the precision is assumed to be 1.

.*n*	For **d**, **i**, **o**, **u**, **x**, **X** types then at least n digits are printed (padding with zeros if necessary). For **f**, **e**, **E** types specifies the number of digits after the decimal point. For **g** or **G** types specifies the number of significant digits to print. For **s** type specifies the maximum number of characters to print.

Modifier:
A modifier changes the way a conversion specifier type is interpreted.

[modifier]	[type]	Effect
h	**d**, **i**, **o**, **u**, **x**, **X**	Value is first converted to a short int or unsigned short i nt.
h	**n**	Specifies that the pointer points to a short int.
l	**d**, **i**, **o**, **u**, **x**, **X**	Value is first converted to a long int or unsigned long int .
l	**n**	Specifies that the pointer points to a long int.
L	**e**, **E**, **f**, **g**, **G**	Value is first converted to a long double.

Conversion specifier type:
The conversion specifier specifies what type the argument is to be treated as.

[type]	Output
d, **i**	Type **signed int**.
o	Type **unsigned int** printed in octal.
u	Type **unsigned int** printed in decimal.
x	Type **unsigned int** printed in hexadecimal as dddd using a, b, c, d, e, f.
X	Type **unsigned int** printed in hexadecimal as dddd using A, B, C, D, E, F.
f	Type **double** printed as [-]ddd.ddd.
e, **E**	Type **double** printed as [-]d.dddeñdd where there is one digit printed before the decimal (zero only if the value is zero). The exponent contains at least two digits. If type is E then the exponent is printed with a capital E.
g, **G**	Type **double** printed as type e or E if the exponent is less than -4 or greater than or equal to the precision. Otherwise printed as type f. Trailing zeros are removed. Decimal point character appears only if there is a nonzero decimal digit.
c	Type **char**. Single character is printed.
s	Type pointer to array. String is printed according to precision (no precision prints entire string).
p	Prints the value of a pointer (the memory location it holds).
n	The argument must be a pointer to an **int**. Stores the number of characters printed thus far in the int. No characters are printed.

% A **%** sign is printed.

The number of characters printed are returned. If an error occurred, -1 is returned.

2.12.4.2 ..scanf Functions

Declarations:

```
int fscanf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

The ..scanf functions provide a means to input formatted information from a stream.

fscanf reads formatted input from a stream
scanf reads formatted input from stdin
sscanf reads formatted input from a string

These functions take input in a manner that is specified by the format argument and store each input field into the following arguments in a left to right fashion.

Each input field is specified in the format string with a conversion specifier which specifies how the input is to be stored in the appropriate variable. Other characters in the format string specify characters that must be matched from the input, but are not stored in any of the following arguments. If the input does not match then the function stops scanning and returns. A whitespace character may match with any whitespace character (space, tab, carriage return, new line, vertical tab, or formfeed) or the next incompatible character.

An input field is specified with a conversion specifer which begins with the **%** character. After the **%** character come the following in this order:

[*]	Assignment suppressor (optional).
[width]	Defines the maximum number of characters to read (optional).
[modifier]	Overrides the size (type) of the argument (optional).
[type]	The type of conversion to be applied (required).

Assignment suppressor:
Causes the input field to be scanned but not stored in a variable.

Width:
The maximum width of the field is specified here with a decimal value. If the input is smaller than the width specifier (i.e. it reaches a nonconvertible character), then what was read thus far is converted and stored in the variable.

Modifier:
A modifier changes the way a conversion specifier type is interpreted.

[modifier]	[type]	Effect
h	**d**, **i**, **o**, **u**, **x**	The argument is a **short int** or **unsigned short int**.< /td>
h	**n**	Specifies that the pointer points to a **short int**.
l	**d**, **i**, **o**, **u**, **x**	The argument is a **long int** or **unsigned long int** .
l	**n**	Specifies that the pointer points to a **long int**.
l	**e**, **f**, **g**	The argument is a **double**.
L	**e**, **f**, **g**	The argument is a **long double**.

Conversion specifier type:
The conversion specifier specifies what type the argument is. It also controls what a valid convertible character is (what kind of characters it can read so it can convert to something compatible).

[type]	Input
d	Type **signed int** represented in base 10. Digits 0 through 9 and the sign (+ or -).
i	Type **signed int**. The base (radix) is dependent on the first two characters. If the first character is a digit from 1 to 9, then it is base 10. If the first digit is a zero and the second digit is a digit from 1 to 7, then it is base 8 (octal). If the first digit is a zero and the second character is an x or X, then it is base 16 (hexadecimal).
o	Type **unsigned int**. The input must be in base 8 (octal). Digits 0 through 7 only.
u	Type **unsigned int**. The input must be in base 10 (decimal). Digits 0 through 9 only.
x, X	Type **unsigned int**. The input must be in base 16 (hexadecimal). Digits 0 through 9 or A through Z or a through z. The characters 0x or 0X may be optionally prefixed to the value.
e, **E**, **f**, **g**, **G**	Type **float**. Begins with an optional sign. Then one or more digits, followed by an optional decimal-point and decimal value. Finally ended with an optional signed exponent value designated with an e or E.
s	Type character array. Inputs a sequence of non-whitespace characters (space, tab, carriage return, new line, vertical tab, or formfeed). The array must be large enough to hold the sequence plus a null character appended to the end.

[...]	Type character array. Allows a search set of characters. Allows input of only those character encapsulated in the brackets (the scanset). If the first character is a carrot (^), then the scanset is inverted and allows any ASCII character except those specified between the brackets. On some systems a range can be specified with the dash character (-). By specifying the beginning character, a dash, and an ending character a range of characters can be included in the scanset. A null character is appended to the end of the array.
c	Type character array. Inputs the number of characters specified in the width field. If no width field is specified, then 1 is assumed. No null character is appended to the array.
p	Pointer to a pointer. Inputs a memory address in the same fashion of the **%p** type produced by the printf function.
n	The argument must be a pointer to an **int**. Stores the number of characters read thus far in the **int**. No characters are read from the input stream.
%	Requires a matching **%** sign from the input.

Reading an input field (designated with a conversion specifier) ends when an incompatible character is met, or the width field is satisfied.

On success the number of input fields converted and stored are returned. If an input failure occurred, then EOF is returned.

2.12.5 Character I/O Functions

2.12.5.1 fgetc

Declaration:

```
int fgetc(FILE *stream);
```

Gets the next character (an **unsigned char**) from the specified stream and advances the position indicator for the stream.

On success the character is returned. If the end-of-file is encountered, then **EOF** is returned and the end-of-file indicator is set. If an error occurs then the error indicator for the stream is set and **EOF** is returned.

2.12.5.2 fgets

Declaration:

```
char *fgets(char *str, int n, FILE *stream);
```

Reads a line from the specified stream and stores it into the string pointed to by *str*. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. The newline character is copied to the string. A null character is appended to the end of the string.

On success a pointer to the string is returned. On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.

2.12.5.3 fputc

Declaration:

```
int fputc(int char, FILE *stream);
```

Writes a character (an **unsigned char**) specified by the argument *char* to the specified stream and advances the position indicator for the stream.

On success the character is returned. If an error occurs, the error indicator for the stream is set and **EOF** is returned.

2.12.5.4 fputs

Declaration:

```
int fputs(const char *str, FILE *stream);
```

Writes a string to the specified stream up to but not including the null character.

On success a nonnegative value is returned. On error **EOF** is returned.

2.12.5.5 getc

Declaration:

```
int getc(FILE *stream);
```

Gets the next character (an **unsigned char**) from the specified stream and advances the position indicator for the stream.

This may be a macro version of **fgetc**.

On success the character is returned. If the end-of-file is encountered, then **EOF** is returned and the end-of-file indicator is set. If an error occurs then the error indicator for the stream is set and **EOF** is returned.

2.12.5.6 getchar

Declaration:

```
int getchar(void);
```

Gets a character (an **unsigned char**) from **stdin**.

On success the character is returned. If the end-of-file is encountered, then **EOF** is returned and the end-of-file indicator is set. If an error occurs then the error indicator for the stream is set and **EOF** is returned.

2.12.5.7 gets

Declaration:

```
char *gets(char *str);
```

Reads a line from **stdin** and stores it into the string pointed to by *str*. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first. The newline character is not copied to the string. A null character is appended to the end of the string.

On success a pointer to the string is returned. On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.

2.12.5.8 putc

Declaration:

```
int putc(int char, FILE *stream);
```

Writes a character (an **unsigned char**) specified by the argument *char* to the specified stream and advances the position indicator for the stream.

This may be a macro version of **fputc**.

On success the character is returned. If an error occurs, the error indicator for the stream is set and **EOF** is returned.

2.12.5.9 putchar

Declaration:

```
int putchar(int char);
```

Writes a character (an **unsigned char**) specified by the argument *char* to **stdout**.

On success the character is returned. If an error occurs, the error indicator for the stream is set and **EOF** is returned.

2.12.5.10 puts

Declaration:

```
int puts(const char *str);
```

Writes a string to **stdout** up to but not including the null character. A newline character is appended to the output.

On success a nonnegative value is returned. On error **EOF** is returned.

2.12.5.11 ungetc

Declaration:

```
int ungetc(int char, FILE *stream);
```

Pushes the character *char* (an **unsigned char**) onto the specified stream so that the this is the next character read. The functions **fseek**, **fsetpos**, and **rewind** discard any characters pushed onto the stream.

Multiple characters pushed onto the stream are read in a FIFO manner (first in, first out).

On success the character pushed is returned. On error **EOF** is returned.

2.12.7 Error Functions

2.12.7.1 perror

Declaration:

```
void perror(const char *str);
```

Prints a descriptive error message to stderr. First the string *str* is printed followed by a colon then a space. Then an error message based on the current setting of the variable **errno** is printed.

Previous Section 2.11 stddef.h | Table of Contents | Index | Next Section 2.13 stdlib.h

Previous Section 2.12 stdio.h | Table of Contents | Index | Next Section 2.14 string.h

2.13 stdlib.h

The stdlib header defines several general operation functions and macros.

Macros:

```
NULL
EXIT_FAILURE
EXIT_SUCCESS
RAND_MAX
MB_CUR_MAX
```

Variables:

```
typedef size_t
typedef wchar_t
struct div_t
struct ldiv_t
```

Functions:

```
abort();
abs();
atexit();
atof();
atoi();
atol();
bsearch();
calloc();
div();
exit();
free();
getenv();
labs();
ldiv();
malloc();
```

```
mblen();
mbstowcs();
mbtowc();
qsort();
rand();
realloc();
srand();
strtod();
strtol();
strtoul();
system();
wcstombs();
wctomb();
```

2.13.1 Variables and Definitions

size_t is the unsigned integer result of the **sizeof** keyword.
wchar_t is an integer type of the size of a wide character constant.
div_t is the structure returned by the **div** function.
ldiv_t is the structure returned by the **ldiv** function.

NULL is the value of a null pointer constant.
EXIT_FAILURE and **EXIT_SUCCESS** are values for the exit function to return termination status.
RAND_MAX is the maximum value returned by the rand function.
MB_CUR_MAX is the maximum number of bytes in a multibyte character set which cannot be larger than **MB_LEN_MAX**.

2.13.2 String Functions

2.13.2.1 atof

Declaration:

```
double atof(const char *str);
```

The string pointed to by the argument *str* is converted to a floating-point number (type **double**). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign, a string of digits with an optional decimal character, and an optional **e** or **E** followed by a optionally signed exponent. Conversion stops when the first unrecognized character is reached.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of range of the type double, then **HUGE_VAL** is returned with the appropriate sign and **ERANGE** is stored in the variable **errno**. If the value is too small to be returned in the type **double**, then zero is returned and **ERANGE** is stored in the variable **errno**.

2.13.2.2 atoi

Declaration:

```
int atoi(const char *str);
```

The string pointed to by the argument *str* is converted to an integer (type **int**). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

On success the converted number is returned. If the number cannot be converted, then **0** is returned.

2.13.2.3 atol

Declaration:

```
long int atol(const char *str);
```

The string pointed to by the argument *str* is converted to a long integer (type **long int**). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

On success the converted number is returned. If the number cannot be converted, then **0** is returned.

2.13.2.4 strtod

Declaration:

```
double strtod(const char *str, char **endptr);
```

The string pointed to by the argument *str* is converted to a floating-point number (type **double**). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign, a string of digits with an optional decimal character, and an

optional **e** or **E** followed by a optionally signed exponent. Conversion stops when the first unrecognized character is reached.

The argument *endptr* is a pointer to a pointer. The address of the character that stopped the scan is stored in the pointer that *endptr* points to.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of range of the type double, then **HUGE_VAL** is returned with the appropriate sign and **ERANGE** is stored in the variable **errno**. If the value is too small to be returned in the type **double**, then zero is returned and **ERANGE** is stored in the variable **errno**.

2.13.2.5 strtol

Declaration:

```
long int strtol(const char *str, char **endptr, int base);
```

The string pointed to by the argument *str* is converted to a long integer (type **long int**). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

If the *base* (radix) argument is zero, then the conversion is dependent on the first two characters. If the first character is a digit from 1 to 9, then it is base 10. If the first digit is a zero and the second digit is a digit from 1 to 7, then it is base 8 (octal). If the first digit is a zero and the second character is an x or X, then it is base 16 (hexadecimal).

If the *base* argument is from 2 to 36, then that base (radix) is used and any characters that fall outside of that base definition are considered unconvertible. For base 11 to 36, the characters A to Z (or a to z) are used. If the base is 16, then the characters 0x or 0X may precede the number.

The argument *endptr* is a pointer to a pointer. The address of the character that stopped the scan is stored in the pointer that *endptr* points to.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of the range of the type **long int**, then **LONG_MAX** or **LONG_MIN** is returned with the sign of the correct value and **ERANGE** is stored in the variable **errno**.

2.13.2.6 strtoul

Declaration:

```
unsigned long int strtoul(const char *str, char **endptr, int
base);
```

The string pointed to by the argument *str* is converted to an unsigned long integer (type **unsigned long int**). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

If the *base* (radix) argument is zero, then the conversion is dependent on the first two characters. If the first character is a digit from 1 to 9, then it is base 10. If the first digit is a zero and the second digit is a digit from 1 to 7, then it is base 8 (octal). If the first digit is a zero and the second character is an x or X, then it is base 16 (hexadecimal).

If the *base* argument is from 2 to 36, then that base (radix) is used and any characters that fall outside of that base definition are considered unconvertible. For base 11 to 36, the characters A to Z (or a to z) are used. If the *base* is 16, then the characters 0x or 0X may precede the number.

The argument *endptr* is a pointer to a pointer. The address of the character that stopped the scan is stored in the pointer that *endptr* points to.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of the range of the type **unsigned long int**, then **ULONG_MAX** is returned and **ERANGE** is stored in the variable **errno**.

2.13.3 Memory Functions

2.13.3.1 calloc

Declaration:

```
void *calloc(size_t nitems, size_t size);
```

Allocates the requested memory and returns a pointer to it. The requested size is *nitems* each *size* bytes long (total memory requested is nitems*size). The space is initialized to all zero bits.

On success a pointer to the requested space is returned. On failure a null pointer is returned.

2.13.3.2 free

Declaration:

```
void free(void *ptr);
```

Deallocates the memory previously allocated by a call to **calloc**, **malloc**, or **realloc**. The argument *ptr* points to the space that was previously allocated. If *ptr* points to a memory block that was not allocated with **calloc**, **malloc**, or **realloc**, or is a space that has been deallocated, then the result is undefined.

No value is returned.

2.13.3.3 malloc

Declaration:

```
void *malloc(size_t size);
```

Allocates the requested memory and returns a pointer to it. The requested size is *size* bytes. The value of the space is indeterminate.

On success a pointer to the requested space is returned. On failure a null pointer is returned.

2.13.3.4 realloc

Declaration:

```
void *realloc(void *ptr, size_t size);
```

Attempts to resize the memory block pointed to by *ptr* that was previously allocated with a call to **malloc** or **calloc**. The contents pointed to by *ptr* are unchanged. If the value of *size* is greater than the previous size of the block, then the additional bytes have an undeterminate value. If the value of *size* is less than the previous size of the block, then the difference of bytes at the end of the block are freed. If ptr is null, then it behaves like **malloc**. If *ptr* points to a memory block that was not allocated with **calloc** or **malloc**, or is a space that has been deallocated, then the result is undefined. If the new space cannot be allocated, then the contents pointed to by *ptr* are unchanged. If size is zero, then the memory block is completely freed.

On success a pointer to the memory block is returned (which may be in a different location as before). On failure or if size is zero, a null pointer is returned.

2.13.4 Environment Functions

2.13.4.1 abort

Declaration:

```
void abort(void);
```

Causes an abnormal program termination. Raises the **SIGABRT** signal and an unsuccessful termination status is returned to the environment. Whether or not open streams are closed is implementation-defined.

No return is possible.

2.13.4.2 atexit

Declaration:

```
int atexit(void (*func)(void));
```

Causes the specified function to be called when the program terminates normally. At least 32 functions can be registered to be called when the program terminates. They are called in a last-in, first-out basis (the last function registered is called first).

On success zero is returned. On failure a nonzero value is returned.

2.13.4.3 exit

Declaration:

```
void exit(int status);
```

Causes the program to terminate normally. First the functions registered by atexit are called, then all open streams are flushed and closed, and all temporary files opened with tmpfile are removed. The value of *status* is returned to the environment. If *status* is **EXIT_SUCCESS**, then this signifies a successful termination. If *status* is **EXIT_FAILURE**, then this signifies an unsuccessful termination. All other values are implementation-defined.

No return is possible.

2.13.4.4 getenv

Declaration:

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

Searches for the environment string pointed to by *name* and returns the associated value to the string. This returned value should not be written to.

If the string is found, then a pointer to the string's associated value is returned. If the string is not found, then a null pointer is returned.

2.13.4.5 system

Declaration:

```
int system(const char *string);
```

The command specified by string is passed to the host environment to be executed by the command processor. A null pointer can be used to inquire whether or not the command processor exists.

If string is a null pointer and the command processor exists, then zero is returned. All other return values are implementation-defined.

2.13.5 Searching and Sorting Functions

2.13.5.1 bsearch

Declaration:

```
void *bsearch(const void *key, const void *base, size_t nitems,
size_t size, int (*compar)(const void *, const void *));
```

Performs a binary search. The beginning of the array is pointed to by *base*. It searches for an element equal to that pointed to by *key*. The array is *nitems* long with each element in the array *size* bytes long.

The method of comparing is specified by the *compar* function. This function takes two arguments, the first is the key pointer and the second is the current element in the array being compared. This function must return less than zero if the compared value is less than the specified key. It must return zero if the compared value is equal to the specified key. It must return greater than zero if the compared value is

greater than the specified key.

The array must be arranged so that elements that compare less than key are first, elements that equal key are next, and elements that are greater than key are last.

If a match is found, a pointer to this match is returned. Otherwise a null pointer is returned. If multiple matching keys are found, which key is returned is unspecified.

2.13.5.2 qsort

Declaration:

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)
(const void *, const void*));
```

Sorts an array. The beginning of the array is pointed to by *base*. The array is *nitems* long with each element in the array *size* bytes long.

The elements are sorted in ascending order according to the *compar* function. This function takes two arguments. These arguments are two elements being compared. This function must return less than zero if the first argument is less than the second. It must return zero if the first argument is equal to the second. It must return greater than zero if the first argument is greater than the second.

If multiple elements are equal, the order they are sorted in the array is unspecified.

No value is returned.

Example:

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

int main(void)
{
  char string_array[10][50]={"John",
                             "Jane",
                             "Mary",
                             "Rogery",
                             "Dave",
                             "Paul",
```

```
                                "Beavis",
                                "Astro",
                                "George",
                                "Elroy"};

  /* Sort the list */
  qsort(string_array,10,50,strcmp);

  /* Search for the item "Elroy" and print it */
  printf("%s",bsearch("Elroy",string_array,10,50,strcmp));

  return 0;
}
```

2.13.6 Math Functions

2.13.6.1 abs

Declaration:

```
int abs(int x);
```

Returns the absolute value of *x*. Note that in two's compliment that the most maximum number cannot be represented as a positive number. The result in this case is undefined.

The absolute value is returned.

2.13.6.2 div

Declaration:

```
div_t div(int numer, int denom);
```

Divides *numer* (numerator) by *denom* (denominator). The result is stored in the structure **`div_t`** which has two members:

```
int qout;
int rem;
```

Where *quot* is the quotient and *rem* is the remainder. In the case of inexact division, *quot* is rounded

down to the nearest integer. The value *numer* is equal to **`quot * denom + rem`**.

The value of the division is returned in the structure.

2.13.6.3 labs

Declaration:

```
long int labs(long int x);
```

Returns the absolute value of *x*. Not that in two's compliment that the most maximum number cannot be represented as a positive number. The result in this case is undefined.

The absolute value is returned.

2.13.6.4 ldiv

Declaration:

```
ldiv_t ldiv(long int numer, long int denom);
```

Divides *numer* (numerator) by *denom* (denominator). The result is stored in the structure **`ldiv_t`** which has two members:

```
long int qout;
long int rem;
```

Where *quot* is the quotient and *rem* is the remainder. In the case of inexact division, *quot* is rounded down to the nearest integer. The value *numer* is equal to **`quot * denom + rem`**.

The value of the division is returned in the structure.

2.13.6.5 rand

Declaration:

```
int rand(void);
```

Returns a pseudo-random number in the range of **`0`** to **`RAND_MAX`**.

The random number is returned.

2.13.6.6 srand

Declaration:

```
void srand(unsigned int seed);
```

This function seeds the random number generator used by the function **rand**. Seeding **srand** with the same seed will cause **rand** to return the same sequence of pseudo-random numbers. If **srand** is not called, **rand** acts as if **srand(1)** has been called.

No value is returned.

2.13.7 Multibyte Functions

The behavior of the multibyte functions are affected by the setting of **LC_CTYPE** in the location settings.

2.13.7.1 mblen

Declaration:

```
int mblen(const char *str, size_t n);
```

Returns the length of a multibyte character pointed to by the argument *str*. At most *n* bytes will be examined.

If *str* is a null pointer, then zero is returned if multibyte characters are not state-dependent (shift state). Otherwise a nonzero value is returned if multibyte character are state-dependent.

If *str* is not null, then the number of bytes that are contained in the multibyte character pointed to by *str* are returned. Zero is returned if str points to a null character. A value of -1 is returned if *str* does not point to a valid multibyte character.

2.13.7.2 mbstowcs

Declaration:

```
size_t mbstowcs(schar_t *pwcs, const char *str, size_t n);
```

Converts the string of multibyte characters pointed to by the argument *str* to the array pointed to by *pwcs*. It stores no more than *n* values into the array. Conversion stops when it reaches the null character or *n* values have been stored. The null character is stored in the array as zero but is not counted in the return value.

If an invalid multibyte character is reached, then the value -1 is returned. Otherwise the number of values stored in the array is returned not including the terminating zero character.

2.13.7.3 mbtowc

Declaration:

```
int mbtowc(whcar_t *pwc, const char *str, size_t n);
```

Examines the multibyte character pointed to by the argument *str*. The value is converted and stored in the argument *pwc* if *pwc* is not null. It scans at most *n* bytes.

If *str* is a null pointer, then zero is returned if multibyte characters are not state-dependent (shift state). Otherwise a nonzero value is returned if multibyte character are state-dependent.

If *str* is not null, then the number of bytes that are contained in the multibyte character pointed to by str are returned. Zero is returned if *str* points to a null character. A value of -1 is returned if *str* does not point to a valid multibyte character.

2.13.7.4 wcstombs

Declaration:

```
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n);
```

Converts the codes stored in the array *pwcs* to multibyte characters and stores them in the string *str*. It copies at most *n* bytes to the string. If a multibyte character overflows the *n* constriction, then none of that multibyte character's bytes are copied. Conversion stops when it reaches the null character or *n* bytes have been written to the string. The null character is stored in the string, but is not counted in the return value.

If an invalid code is reached, the value -1 is returned. Otherwise the number of bytes stored in the string is returned not including the terminating null character.

2.13.7.5 wctomb

Declaration:

```
int wctomb(char *str, wchar_t wchar);
```

Examines the code which corresponds to a multibyte character given by the argument *wchar*. The code is converted to a multibyte character and stored into the string pointed to by the argument *str* if *str* is not null.

If *str* is a null pointer, then zero is returned if multibyte characters are not state-dependent (shift state). Otherwise a nonzero value is returned if multibyte character are state-dependent.

If *str* is not null, then the number of bytes that are contained in the multibyte character wchar are returned. A value of -1 is returned if wchar is not a valid multibyte character.

Previous Section 2.12 stdio.h | Table of Contents | Index | Next Section 2.14 string.h

Previous Section 2.13 stdlib.h | Table of Contents | Index | Next Section 2.15 time.h

2.14 string.h

The string header provides many functions useful for manipulating strings (character arrays).

Macros:

```
NULL
```

Variables:

```
typedef size_t
```

Functions:

```
memchr();
memcmp();
memcpy();
memmove();
memset();
strcat();
strncat();
strchr();
strcmp();
strncmp();
strcoll();
strcpy();
strncpy();
strcspn();
strerror();
strlen();
strpbrk();
strrchr();
strspn();
strstr();
strtok();
strxfrm();
```

2.14.1 Variables and Definitions

size_t is the unsigned integer result of the sizeof keyword.
NULL is the value of a null pointer constant.

2.14.2 memchr

Declaration:

```
void *memchr(const void *str, int c, size_t n);
```

Searches for the first occurrence of the character *c* (an **unsigned char**) in the first *n* bytes of the string pointed to by the argument *str*.

Returns a pointer pointing to the first matching character, or null if no match was found.

2.14.3 memcmp

Declaration:

```
int memcmp(const void *str1, const void *str2, size_t n);
```

Compares the first *n* bytes of *str1* and *str2*. Does not stop comparing even after the null character (it always checks *n* characters).

Returns zero if the first *n* bytes of *str1* and *str2* are equal. Returns less than zero or greater than zero if *str1* is less than or greater than *str2* respectively.

2.14.4 memcpy

Declaration:

```
void *memcpy(void *str1, const void *str2, size_t n);
```

Copies n characters from *str2* to *str1*. If *str1* and *str2* overlap the behavior is undefined.

Returns the argument *str1*.

2.14.5 memmove

Declaration:

```
void *memmove(void *str1, const void *str2, size_t n);
```

Copies *n* characters from *str2* to *str1*. If *str1* and *str2* overlap the information is first completely read from *str1* and then written to *str2* so that the characters are copied correctly.

Returns the argument *str1*.

2.14.6 memset

Declaration:

```
void *memset(void *str, int c, size_t n);
```

Copies the character *c* (an **unsigned char**) to the first *n* characters of the string pointed to by the argument *str*.

The argument *str* is returned.

2.14.7 strcat

Declaration:

```
char *strcat(char *str1, const char *str2);
```

Appends the string pointed to by *str2* to the end of the string pointed to by *str1*. The terminating null character of *str1* is overwritten. Copying stops once the terminating null character of *str2* is copied. If overlapping occurs, the result is undefined.

The argument *str1* is returned.

2.14.8 strncat

Declaration:

```
char *strncat(char *str1, const char *str2, size_t n);
```

Appends the string pointed to by *str2* to the end of the string pointed to by *str1* up to *n* characters long. The terminating null character of *str1* is overwritten. Copying stops once *n* characters are copied or the terminating null character of *str2* is copied. A terminating null character is always appended to *str1*. If overlapping occurs, the result is undefined.

The argument *str1* is returned.

2.14.9 strchr

Declaration:

```
char *strchr(const char *str, int c);
```

Searches for the first occurrence of the character *c* (an unsigned char) in the string pointed to by the argument *str*. The terminating null character is considered to be part of the string.

Returns a pointer pointing to the first matching character, or null if no match was found.

2.14.10 strcmp

Declaration:

```
int strcmp(const char *str1, const char *str2);
```

Compares the string pointed to by *str1* to the string pointed to by *str2*.

Returns zero if *str1* and *str2* are equal. Returns less than zero or greater than zero if *str1* is less than or greater than *str2* respectively.

2.14.11 strncmp

Declaration:

```
int strncmp(const char *str1, const char *str2, size_t n);
```

Compares at most the first *n* bytes of *str1* and *str2*. Stops comparing after the null character.

Returns zero if the first *n* bytes (or null terminated length) of *str1* and *str2* are equal. Returns less than zero or greater than zero if *str1* is less than or greater than *str2* respectively.

2.14.12 strcoll

Declaration:

```
int strcoll(const char *str1, const char *str2);
```

Compares string *str1* to *str2*. The result is dependent on the **LC_COLLATE** setting of the location.

Returns zero if *str1* and *str2* are equal. Returns less than zero or greater than zero if *str1* is less than or greater than *str2* respectively.

2.14.13 strcpy

Declaration:

```
char *strcpy(char *str1, const char *str2);
```

Copies the string pointed to by *str2* to *str1*. Copies up to and including the null character of *str2*. If *str1* and *str2* overlap the behavior is undefined.

Returns the argument *str1*.

2.14.14 strncpy

Declaration:

```
char *strncpy(char *str1, const char *str2, size_t n);
```

Copies up to *n* characters from the string pointed to by *str2* to *str1*. Copying stops when *n* characters are copied or the terminating null character in *str2* is reached. If the null character is reached, the null characters are continually copied to *str1* until *n* characters have been copied.

Returns the argument *str1*.

2.14.15 strcspn

Declaration:

```
size_t strcspn(const char *str1, const char *str2);
```

Finds the first sequence of characters in the string *str1* that does not contain any character specified in *str2*.

Returns the length of this first sequence of characters found that do not match with *str2*.

2.14.16 strerror

Declaration:

```
char *strerror(int errnum);
```

Searches an internal array for the error number *errnum* and returns a pointer to an error message string.

Returns a pointer to an error message string.

2.14.17 strlen

Declaration:

```
size_t strlen(const char *str);
```

Computes the length of the string *str* up to but not including the terminating null character.

Returns the number of characters in the string.

2.14.18 strpbrk

Declaration:

```
char *strpbrk(const char *str1, const char *str2);
```

Finds the first character in the string *str1* that matches any character specified in *str2*.

A pointer to the location of this character is returned. A null pointer is returned if no character in *str2* exists in *str1*.

Example:

```
#include<string.h>
```

```
#include<stdio.h>

int main(void)
{
  char string[]="Hi there, Chip!";
  char *string_ptr;

  while((string_ptr=strpbrk(string," "))!=NULL)
    *string_ptr='-';

  printf("New string is \"%s\".\n",string);
  return 0;
}
```

The output should result in every space in the string being converted to a dash (-).

2.14.19 strrchr

Declaration:

```
char *strrchr(const char *str, int c);
```

Searches for the last occurrence of the character *c* (an **unsigned char**) in the string pointed to by the argument *str*. The terminating null character is considered to be part of the string.

Returns a pointer pointing to the last matching character, or null if no match was found.

2.14.20 strspn

Declaration:

```
size_t strspn(const char *str1, const char *str2);
```

Finds the first sequence of characters in the string *str1* that contains any character specified in *str2*.

Returns the length of this first sequence of characters found that match with *str2*.

Example:

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

```
int main(void)
{
  char string[]="7803 Elm St.";

  printf("The number length is %d.\n",strspn
(string,"1234567890"));

  return 0;
}
```

The output should be: The number length is 4.

2.14.21 strstr

Declaration:

```
char *strstr(const char *str1, const char *str2);
```

Finds the first occurrence of the entire string *str2* (not including the terminating null character) which appears in the string *str1*.

Returns a pointer to the first occurrence of *str2* in *str1*. If no match was found, then a null pointer is returned. If *str2* points to a string of zero length, then the argument *str1* is returned.

2.14.22 strtok

Declaration:

```
char *strtok(char *str1, const char *str2);
```

Breaks string *str1* into a series of tokens. If *str1* and *str2* are not null, then the following search sequence begins. The first character in *str1* that does not occur in *str2* is found. If *str1* consists entirely of characters specified in *str2*, then no tokens exist and a null pointer is returned. If this character is found, then this marks the beginning of the first token. It then begins searching for the next character after that which is contained in *str2*. If this character is not found, then the current token extends to the end of *str1*. If the character is found, then it is overwritten by a null character, which terminates the current token. The function then saves the following position internally and returns.

Subsequent calls with a null pointer for *str1* will cause the previous position saved to be restored and begins searching from that point. Subsequent calls may use a different value for *str2* each time.

Returns a pointer to the first token in *str1*. If no token is found then a null pointer is returned.

Example:

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

int main(void)
{
  char search_string[]="Woody Norm Cliff";
  char *array[50];
  int loop;

  array[0]=strtok(search_string," ");
  if(array[0]==NULL)
   {
    printf("No test to search.\n");
    exit(0);
   }

  for(loop=1;loop<50;loop++)
   {
    array[loop]=strtok(NULL," ");
    if(array[loop]==NULL)
      break;
   }

  for(loop=0;loop<50;loop++)
   {
    if(array[loop]==NULL)
      break;
    printf("Item #%d is %s.\n",loop,array[loop]);
   }

  return 0;
}
```

This program replaces each space into a null character and stores a pointer to each substring into the array. It then prints out each item.

2.14.23 strxfrm

Declaration:

```
size_t strxfrm(char *str1, const char *str2, size_t n);
```

Transforms the string *str2* and places the result into *str1*. It copies at most *n* characters into *str1* including the null terminating character. The transformation occurs such that **strcmp** applied to two separate converted strings returns the same value as **strcoll** applied to the same two strings. If overlapping occurs, the result is undefined.

Returns the length of the transformed string (not including the null character).

Previous Section 2.13 stdlib.h | Table of Contents | Index | Next Section 2.15 time.h

Previous Section 2.14 string.h | Table of Contents | Index | Next Section Index

2.15 time.h

The time header provides several functions useful for reading and converting the current time and date. Some functions behavior is defined by the **LC_TIME** category of the location setting.

Macros:

```
NULL
CLOCKS_PER_SEC
```

Variables:

```
typedef size_t
typedef clock_t
typedef size_t
struct tm
```

Functions:

```
asctime();
clock();
ctime();
difftime();
gmtime();
localtime();
mktime();
strftime();
time();
```

2.15.1 Variables and Definitions

NULL is the value of a null pointer constant.
CLOCKS_PER_SEC is the number of processor clocks per second.

size_t is the unsigned integer result of the sizeof keyword.

clock_t is a type suitable for storing the processor time.
time_t is a type suitable for storing the calendar time.

struct tm is a structure used to hold the time and date. Its members are as follows:

```
int tm_sec;    /* seconds after the minute (0 to 61) */
int tm_min;    /* minutes after the hour (0 to 59) */
int tm_hour;   /* hours since midnight (0 to 23) */
int tm_mday;   /* day of the month (1 to 31) */
int tm_mon;    /* months since January (0 to 11) */
int tm_year;   /* years since 1900 */
int tm_wday;   /* days since Sunday (0 to 6 Sunday=0)
*/
int tm_yday;   /* days since January 1 (0 to 365) */
int tm_isdst;  /* Daylight Savings Time */
```

If **tm_isdst** is zero, then Daylight Savings Time is not in effect. If it is a positive value, then Daylight Savings Time is in effect. If it is negative, then the function using it is requested to attempt to calculate whether or not Daylight Savings Time is in effect for the given time.

Note that **tm_sec** may go as high as 61 to allow for up to two leap seconds.

2.15.2 asctime

Declaration:

```
char *asctime(const struct tm *timeptr);
```

Returns a pointer to a string which represents the day and time of the structure *timeptr*. The string is in the following format:

```
DDD MMM dd hh:mm:ss YYYY
```

DDD	Day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat)
MMM	Month of the year (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)
dd	Day of the month (1,...,31)
hh	Hour (0,...,23)
mm	Minute (0,...,59)
ss	Second (0,...,59)
YYYY	Year

The string is terminated with a newline character and a null character. The string is always 26 characters long (including the terminating newline and null characters).

A pointer to the string is returned.

Example:

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

int main(void)
{
  time_t timer;

  timer=time(NULL);
  printf("The current time is %s.\n",asctime(localtime
(&timer)));
  return 0;
}
```

2.15.3 clock

Declaration:

```
clock_t clock(void);
```

Returns the processor clock time used since the beginning of an implementation-defined era (normally the beginning of the program). The returned value divided by **CLOCKS_PER_SEC** results in the number of seconds. If the value is unavailable, then -1 is returned.

Example:

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

int main(void)
{
  clock_t ticks1, ticks2;

  ticks1=clock();
  ticks2=ticks1;
  while((ticks2/CLOCKS_PER_SEC-ticks1/CLOCKS_PER_SEC)<1)
```

```
    ticks2=clock();

  printf("Took %ld ticks to wait one second.\n",ticks2-
ticks1);
  printf("This value should be the same as CLOCKS_PER_SEC
which is %ld.\n",CLOCKS_PER_SEC);
  return 0;
}
```

2.15.4 ctime

Declaration:

```
char *ctime(const time_t *timer);
```

Returns a string representing the localtime based on the argument *timer*. This is equivalent to:

```
asctime(locatime(timer));
```

The returned string is in the following format:

```
DDD MMM dd hh:mm:ss YYYY
```

DDD	Day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat)
MMM	Month of the year (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)
dd	Day of the month (1,...,31)
hh	Hour (0,...,23)
mm	Minute (0,...,59)
ss	Second (0,...,59)
YYYY	Year

The string is terminated with a newline character and a null character. The string is always 26 characters long (including the terminating newline and null characters).

A pointer to the string is returned.

2.15.5 difftime

Declaration:

double difftime(time_t *time1*, **time_t** *time2*);

Calculates the difference of seconds between *time1* and *time2* (time1-time2).

Returns the number of seconds.

2.15.6 gmtime

Declaration:

**struct tm *gmtime(const time_t *****timer);*

The value of timer is broken up into the structure `tm` *and expressed in Coordinated Universal Time (UTC) also known as Greenwich Mean Time (GMT).*

A pointer to the structure is returned. A null pointer is returned if UTC is not available.

2.15.7 localtime

Declaration:

```
struct tm *localtime(const time_t *timer);
```

The value of timer is broken up into the structure `tm` *and expressed in the local time zone.*

A pointer to the structure is returned.

Example:

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

int main(void)
{
  time_t timer;

  timer=time(NULL);
  printf("The current time is %s.\n",asctime(localtime
(&timer)));
  return 0;
```

```
}
```

2.15.8 mktime

Declaration:

```
time_t mktime(struct tm *timeptr);
```

Converts the structure pointed to by timeptr into a `time_t` *value according to the local time zone. The values in the structure are not limited to their constraints. If they exceed their bounds, then they are adjusted accordingly so that they fit within their bounds. The original values of* `tm_wday` *(day of the week) and* `tm_yday` *(day of the year) are ignored, but are set correctly after the other values have been constrained.* `tm_mday` *(day of the month) is not corrected until after* `tm_mon` *and* `tm_year` *are corrected.*

After adjustment the structure still represents the same time.

The encoded `time_t` *value is returned. If the calendar time cannot be represented, then -1 is returned.*

Example:

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

/* find out what day of the week is January 1, 2001
  (first day of the 21st century) */

int main(void)
{
  struct tm time_struct;
  char days[7][4]={"Sun", "Mon", "Tue", "Wed", "Thu",
"Fri", "Sat"};

  time_struct.tm_year=2001-1900;
  time_struct.tm_mon=0;
  time_struct.tm_mday=1;
  time_struct.tm_sec=0;
  time_struct.tm_min=0;
  time_struct.tm_hour=0;
  time_struct.tm_isdst=-1;

  if(mktime(&time_struct)==-1)
```

```
    {
     printf("Error getting time.\n");
     exit(0);
    }

  printf("January 1, 2001 is a %s.\n",days[time_struct.
tm_wday]);
  return 0;
}
```

2.15.9 strftime

Declaration:

```
size_t strftime(char *str, size_t maxsize, const char *format,
const struct tm *timeptr);
```

Formats the time represented in the structure timeptr according to the formatting rules defined in format and stored into str. No more than maxsize characters are stored into str (including the terminating null character).

All characters in the format string are copied to the str string, including the terminating null character, except for conversion characters. A conversion character begins with the % sign and is followed by another character which defines a special value that it is to be replaced by.

Conversion Character	What it is replaced by
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	appropriate date and time representation
%d	day of the month (01-31)
%H	hour of the day (00-23)
%I	hour of the day (01-12)
%j	day of the year (001-366)
%m	month of the year (01-12)
%M	minute of the hour (00-59)
%p	AM/PM designator

%S	second of the minute (00-61)
%U	week number of the year where Sunday is the first day of week 1 (00-53)
%w	weekday where Sunday is day 0 (0-6)
%W	week number of the year where Monday is the first day of week 1 (00-53)
%x	appropriate date representation
%X	appropriate time representation
%y	year without century (00-99)
%Y	year with century
%Z	time zone (possibly abbreviated) or no characters if time zone isunavailable
%%	%

Returns the number of characters stored into str not including the terminating null character. On error zero is returned.

2.15.10 time

Declaration:

```
time_t time(time_t *timer);
```

Calculates the current calender time and encodes it into `time_t` format.

The `time_t` value is returned. If timer is not a null pointer, then the value is also stored into the object it points to. If the time is unavailable, then -1 is returned.

Previous Section 2.14 string.h | Table of Contents | Index | Next Section Index

| Table of Contents | Index |

Appendix A

ASCII Chart

Decimal	Octal	Hex	Character
0	0	00	NUL
1	1	01	SOH
2	2	02	STX
3	3	03	ETX
4	4	04	EOT
5	5	05	ENQ
6	6	06	ACK
7	7	07	BEL
8	10	08	BS
9	11	09	HT
10	12	0A	LF
11	13	0B	VT
12	14	0C	FF
13	15	0D	CR
14	16	0E	SO
15	17	0F	SI
16	20	10	DLE
17	21	11	DC1
18	22	12	DC2
19	23	13	DC3
20	24	14	DC4
21	25	15	NAK
22	26	16	SYM
23	27	17	ETB
24	30	18	CAN

Decimal	Octal	Hex	Character
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X

25	31	19	EM	89	131	59	Y
26	32	1A	SUB	90	132	5A	Z
27	33	1B	ESC	91	133	5B	[
28	34	1C	FS	92	134	5C	\
29	35	1D	GS	93	135	5D	]
30	36	1E	RS	94	136	5E	^
31	37	1F	US	95	137	5F	_
32	40	20	SP	96	140	60	`
33	41	21	!	97	141	61	a
34	42	22	"	98	142	62	b
35	43	23	#	99	143	63	c
36	44	24	$	100	144	64	d
37	45	25	%	101	145	65	e
38	46	26	&	102	146	66	f
39	47	27	'	103	147	67	g
40	50	28	(	104	150	68	h
41	51	29	)	105	151	69	i
42	52	2A	*	106	152	6A	j
43	53	2B	+	107	153	6B	k
44	54	2C	,	108	154	6C	l
45	55	2D	-	109	155	6D	m
46	56	2E	.	110	156	6E	n
47	57	2F	/	111	157	6F	o
48	60	30	0	112	160	70	p
49	61	31	1	113	161	71	q
50	62	32	2	114	162	72	r
51	63	33	3	115	163	73	s
52	64	34	4	116	164	74	t
53	65	35	5	117	165	75	u
54	66	36	6	118	166	76	v
55	67	37	7	119	167	77	w
56	70	38	8	120	170	78	x

57	71	39	9	121	171	79	y
58	72	3A	:	122	172	7A	z
59	73	3B	;	123	173	7B	{
60	74	3C	<	124	174	7C	|
61	75	3D	=	125	175	7D	}
62	76	3E	>	126	176	7E	~
63	77	3F	?	127	177	7F	DEL

| Table of Contents | Index |