L

label — Definition

A *label* is an identifier followed by a colon ':' or that follows a **goto** statement. It marks a point within a function to which a **goto** statement can jump.

Cross-references

Standard, §3.1.2.6
The C Programming Language, ed. 2, pp. 65

See Also

goto, name space

labs() — General utility (libc)

Compute the absolute value of a long integer
#include <stdlib.h>
int labs(long *n***);**

labs computes the absolute value of the long integer *n*. The *absolute value* of a number is its distance from zero. This is *n* if *n***>=0**, and -*n* otherwise.

Cross-references

Standard, §4.10.6.3
The C Programming Language, ed. 2, p. 253

See Also

abs, general utilities

Language — Overview

The description of the language, both in the Standard and in this Lexicon, has the following topics, which describe completely the syntax and semantics of the language:

- constant expressions
- conversions
- declarations
- expressions
- external definitions
- lexical elements
- preprocessing
- statements

Each of these topics is introduced by its own Lexicon article.

Implementation of the C Language

The following summarizes how **Let's C** implements the C language.

Identifiers:

Characters allowed: **A-Z**, **a-z**, _, **0-9**
Case sensitive.
Number of significant characters in a variable name:

at compile time:	**128**
at link time:	**16**

Appends '_' to end of external identifiers

Reserved identifiers (keywords):

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

In conformity with the proposed ANSI standard, the keyword **entry** is no longer recognized. The keywords **const** and **volatile** are now recognized, but not implemented. The compiler will produce a warning message if the keyword **volatile** is used with the peephole optimizer.

Data formats (in bits):

char	8
double	64
float	32
int	16
long	32
long double	64
pointer (SMALL model)	16
pointer (LARGE model)	32
short	16
unsigned char	8
unsigned int	16
unsigned long	32
unsigned short	16

float *format:*

IEEE floating point format:
- 1 sign bit
- 8-bit exponent
- 24-bit normalized fraction with hidden bit

IEEE double format:
- 1 sign bit
- 11-bit exponent
- 52-bit fraction

Reserved values:
- +- infinity, -0

All floating-point operations are done as **double**s.
Note that this will change when the ANSI standard is adopted.

Limits:

Maximum bitfield size: 16 bits
Maximum number of **case**s in a **switch**: no formal limit
Maximum block nesting depth: no formal limit
Maximum parentheses nesting depth: no formal limit
Maximum structure size: 64 kilobytes
Maximum array size: 64 kilobytes

Structure name-spaces:

Supports both Berkeley, and Kernighan and Ritchie conventions for structure in union.

Register variables:

Two available for **int**s (SMALL and LARGE models)
Two available for pointers (SMALL model only)

Function linkage:

Return values for **int**s: AX
Return values for **long**s: DX:AX
Return values for SMALL-model pointers: AX
Return values for LARGE-model pointers: DX:AX
Return values for **double**s in DX:AX
Parameters pushed on stack in reverse order, **char**s and **short**s pushed as words, **long**s and pointers pushed as **long**s, structures copied onto stack
Caller must clear parameters off stack
Stack frame linkage is done through SP register

Register usage:

AX:	returned **int**s and SMALL-model pointers
BP:	Frame pointer
DI:	register variable (**int** or SMALL-model pointer)
DX:AX:	returned **long**s and LARGE-model pointers
SI:	register variable (**int** or SMALL-model pointer)

Note that registers not described above (BX, CX, DX, plus DS and ES in LARGE model) may be freely overwritten by code that the compiler generates. Programs that include assembly-language modules should take this into account.

Special features and optimizations:

LEXICON

- Branch optimization is performed: this uses the smallest branch instruction for the required range.
- Unreached code is eliminated.
- The contents of word registers are remembered by a peephole optimizer, to avoid reloading.
- Duplicate instruction sequences are removed.
- Jumps to jumps are eliminated.
- Multiplication and division by constant powers of two are changed to shifts when the results are the same.
- Sequences that can be resolved at compile time are identified and resolved.

Cross-references

Standard, §3.0
The C Programming Language, ed. 2, pp. 191*ff*

See Also

byte ordering, declarations, function calls, keywords, Lexicon, Library, memory allocation, types

LARGE model — Technical information

Intel multi-segment memory model

The i8086/88 microprocessor uses a *segmented architecture*. This means that memory is divided into segments of 64 kilobytes each. No program or data element can exceed that limit.

Intel Corporation has devised a number of memory models for handling segmented memory. **Let's C** implements the two most useful of these: SMALL model and LARGE model.

In LARGE model, pointers consist of an offset and a segment. The address is calculated by left-shifting the segment by four and adding the offset. Thus, LARGE model programs can access up to 1,048,576 bytes (one megabyte) of code and data. Because of the design of the IBM PC and compatibles, however, the practical limit of memory is 640 kilobytes.

In terms of execution, LARGE-model programs are less efficient than SMALL-model programs, but for many purposes the advantages of the expanded address space of the LARGE model outweigh the decreased efficiency.

When the **-VLARGE** option is used with the **cc** command, the object program follows the rules of the LARGE model. When you compile a program with the **-VLARGE** option, **cc** defines the global variable **LARGE** to the C preprocessor. This allows you to use the preprocessor statement **#ifdef LARGE** to flag model-dependent code.

See Also

model, pointer, SMALL model, technical information

LC_ALL — Manifest constant

All locale information
#include <locale.h>

LC_ALL is a manifest constant that is defined in the header **locale.h**. When passed to the function **setlocale**, it queries or sets all information for a given locale. Information obtained with this macro alters the operation of all functions that are affected by the program's locale, as well as the contents of the structure **lconv**. The following lists the functions affected by **LC_ALL**:

Collation
- **strcoll**
- **strxfrm**

ctype
- **isdigit**
- **isxdigit**

Date and time
- **strftime**

Formatted I/O
- **fprintf**
- **fscanf**
- **printf**
- **sprintf**
- **scanf**
- **sscanf**
- **vfprintf**
- **vprintf**
- **vsprintf**

Multibyte characters
- **mblen**
- **mbstowcs**
- **mbtowc**
- **wcstombs**
- **wctomb**

String conversion
- **atof**
- **atoi**
- **atol**
- **strtod**
- **strtol**
- **strtoul**

Cross-reference

Standard, §4.4

See Also

LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, lconv, localization, locale.h, setlocale

LC_COLLATE — Manifest constant

Locale collation information
#include <locale.h>

LC_COLLATE is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets collation information for a given locale.

This information can affect the operation of the functions **strcoll** and **strxfrm**.

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, localization, locale.h, setlocale

LC_CTYPE — Manifest constant

Locale character-handling information
#include <locale.h>

LC_CTYPE is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it sets or queries the character-handling information for a given locale. This information helps determine the action of the functions declared in **ctype.h**, except **isdigit** and **isxdigit**, as well as the multiple-byte character functions **mblen**, **mbstowcs**, **mbtowc**, **wcstombs**, and **wctomb**.

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_MONETARY, LC_NUMERIC, LC_TIME, lconv, localization, locale.h, setlocale

LC_MONETARY — Manifest constant

Locale monetary information
#include <locale.h>

LC_MONETARY is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets the monetary information for a given locale.

It affects all of the fields within the structure **lconv**, except **decimal_point**.

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_CTYPE, LC_NUMERIC, LC_TIME, localization, locale.h, setlocale

LC_NUMERIC — Manifest constant

Locale numeric information
#include <locale.h>

LC_NUMERIC is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets the information for formatting numeric strings.

This information will alter the operation of the following functions:

Formatted I/O
- **fprintf**
- **fscanf**
- **printf**
- **sprintf**
- **scanf**
- **sscanf**
- **vfprintf**
- **vprintf**
- **vsprintf**

String conversion

atof
atoi
atol
strtod
strtol
strtoul

This information also affects the following fields within the structure **lconv**:

decimal_point
thousands_sep
grouping

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_TIME, lconv, localization, locale.h, setlocale

LC_TIME — Manifest constant

Locale time information
#include <locale.h>

LC_TIME is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets the information for formatting time strings.

This information affects the operation of the function **strftime**.

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, lconv, localization, locale.h, setlocale

lconv — Type

Hold monetary conversion information
#include <locale.h>

lconv is a structure that is defined in the header **locale.h**. Its members hold many details needed to format monetary and non-monetary numeric information for a given locale.

To initialize **lconv** for any given locale, use the function **localeconv**. To change any aspect of the locale information being used, use the function **setlocale.**

lconv contains the following fields:

char *currency_symbol
This points to a string that contains the symbol used locally to represent currency, e.g., the '$'. The **C** locale sets this to point to a null string.

char *decimal_point
This points to a string that contains the character used to indicate the decimal point. The **C** locale sets this to point to '.'.

char frac_digits
This is the number of fractional digits that can be displayed in a monetary string. The **C** locale sets this to **CHAR_MAX**.

char grouping
This points to the string that indicates the grouping characteristics for non-monetary amounts. Characters in the string can take the following values:

0	Use previous element for rest of digits
MAX_CHAR	Perform no further grouping
2 through **9**	No. of digits in current group

The **C** locale sets this to **CHAR_MAX**.

char *int_curr_symbol
This points to a string that contains the international currency symbol for the locale, as defined in the publication *ISO 4217 Codes for Representation of Currency and Funds.* The **C** locale sets this to point to a null string.

char *mon_decimal_point
This points to a string that contains the character used to indicate a decimal point in monetary strings. The **C** locale sets this to point to a null string.

char mon_grouping
This points to the string of characters that indicate the grouping characteristics for monetary amounts. Elements can take the following values:

0	Use previous element for rest of digits
MAX_CHAR	Perform no further grouping
2 through **9**	No. of digits in current group

The **C** locale sets this to **CHAR_MAX**.

char *mon_thousands_sep
This points to a string that contains the character used to separate groups of thousands in monetary strings. The **C** locale sets this to point to a null string.

char n_cs_precedes
This indicates whether the symbol that indicates a negative monetary value precedes or follows the numerals in the monetary string. Zero indicates that it follows the numerals and one indicates that it precedes them. The **C** locale sets this to **CHAR_MAX**.

char n_sep_by_space
This indicates whether a space should appear between the symbol that indicates a negative monetary value and the numerals of the monetary string. Zero indicates that it should not appear, and one indicates that it should. The **C** locale sets this to **CHAR_MAX**.

char n_sign_posn
This indicates the position and formatting of the symbol that indicates a negative monetary value, as follows:

0	Set parentheses around numerals and monetary symbol
1	Set negative sign before currency symbol and numerals
2	Set negative sign after currency symbol and numerals
3	Set negative sign immediately before monetary symbol
4	Set negative sign immediately after monetary symbol

The **C** locale sets this to **CHAR_MAX**.

char *negative_sign
This points to a string that contains the character that indicates a negative value in a monetary string. The **C** locale sets this to point to a null string.

char p_cs_precedes
This indicates whether the currency symbol should precede or follow the numerals in the string. Zero indicates that it precedes the digits and one indicates that it follows. The **C** locale sets this to **CHAR_MAX**.

char p_sep_by_space
This indicates whether a space should appear between the monetary symbol and the numerals of the monetary string. Zero indicates that a space should not appear, and one indicates that it should. The **C** locale sets this to **CHAR_MAX**.

char p_sign_posn
This indicates the position and formatting of the symbol that indicates a positive monetary value, as follows:

0	Set parentheses around numerals and monetary symbol
1	Set positive sign before currency symbol and numerals
2	Set positive sign after currency symbol and numerals
3	Set positive sign immediately before monetary symbol
4	Set positive sign immediately after monetary symbol

The **C** locale sets this to **CHAR_MAX**.

char *positive_sign
This points to a string that contains the character that indicates a non-negative value in a monetary string. The **C** locale sets this to point to a null string.

char *thousands_sep
This points to a string that contains the character used to separate groups of thousands. The **C** locale sets this to point to a null string.

Cross-reference

Standard, §4.4, §4.4.2.1

See Also

CHAR_MAX, locale.h, localeconv, localization, setlocale

ldexp() — Mathematics (libm)

Load floating-point number
#include <math.h>
double ldexp(double *number*, **int** *n*);

ldexp returns *number* times two to the *n* power.

See **float.h** for more information on the structure of a floating-point number.

Cross-references

Standard, §4.5.4.3
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, log, log10, mathematics, modf

ldiv() — General utility (libc)

Perform long integer division
#include <stdlib.h>
ldiv_t ldiv(long int *numerator*, **long int** *denominator***);**

ldiv divides *numerator* by *denominator*. It returns a structure of the type **ldiv_t**, which consists of two **long** members, one named **quot** and the other **rem**. **ldiv** writes the quotient into one **long**, and it writes the remainder into the other.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators **/** and **%**, which merely do what the machine implements for divide.

Example

This example selects one random card out of a pack of 52.

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

main(void)
{
      ldiv_t card;

      card = ldiv((unsigned long)time(NULL) % 52, 13L);
      printf("%c%c\n",
            /* note useful addressing for strings */
            "A23456789TJQK"[card.rem],
            "HCDS"[card.quot]);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.6.2
The C Programming Language, ed. 2, p. 253

See Also

/, div, general utilities, ldiv_t

Notes

The Standard includes this function to provide a useful feature of FORTRAN. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **ldiv** is undefined.

ldiv_t — Type

Type returned by ldiv()
#include <stdlib.h>

ldiv_t is a typedef that is declared in the header **stdlib.h** and is the type returned by the function **ldiv**.

ldiv_t is a structure that consists of two **long** members, one named **quot** and the other **rem**. **ldiv** writes its quotient into **quot** and its remainder into **rem**.

Example

For an example of this type in a program, see **ldiv**.

Cross-references

Standard, §4.10.6.2
The C Programming Language, ed. 2, p. 253

See Also

general utilities, integer arithmetic, ldiv, stdlib.h

lexical elements — Overview

A *lexical element* is one of the elements from which a C program is built. It is the smallest unit with which a translator can work. "Lexical" refers to the fact that a program is partitioned into tokens during a translation phase that is usually called "lexical analysis."

A C program is built from the following lexical elements:

constants
header names
identifiers
keywords
operators
preprocessing numbers
punctuators
string literals

Cross-reference

Standard, §3.1

See Also

comment, constant, header name, identifier, keyword, Language, operators, preprocessing number, punctuators, string literal, token

Lexicon — Introduction

The Mark Williams Lexicon is a new approach to documentation of computer software. The Lexicon is designed to improve documentation and eliminate some limitations found in more conventional documentation.

How to Use the Lexicon

The Lexicon consists of one large document that contains entries for every aspect of **Let's C**. You will not have to search through a number of different manuals to find the entry you are looking for.

Every entry in the Lexicon has the same structure. The first line gives the name of the topic being discussed, followed by its type (e.g., **Mathematics**) and, where appropriate, the file in which it is kept.

The next lines briefly describe the item, then give the item's usage, where applicable. These are followed by a brief discussion of the item, and an example.

Cross-references follow. These can be to other entries or to other texts, notably to the ANSI Standard, *The Art of Computer Programming* and the second edition of *The C Programming Language*. Diagnostics and notes, where applicable, conclude each entry.

Internally, the Lexicon has a tree structure. The “root” entry is the present entry, for **Lexicon**. Below this entry comes the set of *Overview* entries. Each Overview entry introduces a group of entries; for example, the Overview entry for **string** introduces all of the string functions and macros, lists them, and gives a lengthy example of how to use them.

Each entry cross-references other entries. These cross-references point up the documentation tree, toward an overview article and, ultimately, to the entry for **Lexicon** itself. They also point down the tree to subordinate entries, and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree.

Use the Lexicon

If, while reading an entry, you encounter a technical term that you do not understand, look it up in the Lexicon. You should find an entry for it. For example, if a function is said to return a data type **float** and you do not know exactly what a **float** is, look it up. You will find it described in full. In this way, you should increase your understanding of **Let’s C**, and make your programming easier and more productive.

libcxs87.lib — Library

Standard library, SMALL model/i8087 only

libcxs87.lib is the archive file that holds the SMALL-model version of the more commonly used C functions, system calls, and compiler run-time support routines.

The routines in this library use the i8087 exclusively. They cannot be run on a computer that does not contain an i8087.

To edit this library or create a table of its contents, use the librarian **mwlib**.

See Also

Library, mwlib

libm — Library

libm is the archive file that holds the mathematics library. For a summary of these routines, see **mathematics** and **extended mathematics**.

libm’s table of contents can be printed and its contents altered with the archiver **mwlib**.

See Also

extended mathematics, Library, mathematics, math.h, mwlib, xmath.h

LIBPATH — Environmental variable

Directories that hold libraries

LIBPATH names the directories that **cc** searches to find the compiler’s executable programs and libraries. **make** also searches these directories for the files **mmacros** and **mactions**.

For example, the command

```
set LIBPATH=\lib;\mwc
```

uses the MS-DOS command **set** to define **LIBPATH** as equalling **\lib;\mwc**. This definition of **LIBPATH** tells **cc** to look for the compiler’s executable files first in directory **lib**, and then in directory **mwc**.

You may wish to write this command into the file **autoexec.bat**, so that **INCDIR** will be set automatically whenever you boot your system.

See Also

cc, environmental variables, make, PATH

limits.h — Header

The header **limits.h** defines a group of macros that set the numerical limits for the translation environment.

The following table gives the macros defined in **limits.h**. Each value given is the macro's minimum maximum: a conforming implementation of C must meet these limits, and may exceed them.

CHAR_BIT
Number of bits in a **char**; must be at least eight.

CHAR_MAX
Largest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MAX**; otherwise, it is equal to the value of the macro **UCHAR_MAX**.

CHAR_MIN
Smallest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MIN**; otherwise, it is zero.

INT_MAX
Largest value representable in an object of type **int**; it must be at least 32,767.

INT_MIN
Smallest value representable in an object of type **int**; it must be at most -32,767.

LONG_MAX
Largest value representable in an object of type **long int**; it must be at least 2,147,483,647.

LONG_MIN
Smallest value representable in an object of type **long int**; it must be at most -2,147,483,647.

MB_LEN_MAX
Largest number of bytes in any multibyte character, for any locale; it must be at least one.

SCHAR_MAX
Largest value representable in an object of type **signed char**; it must be at least 127.

SCHAR_MIN
Smallest value representable in an object of type **signed char**; it must be at most -127.

SHRT_MAX
Largest value representable in an object of type **short int**; it must be at least 32,767.

SHRT_MIN
Smallest value representable in an object of type **short int**; it must be at most -32,767.

UCHAR_MAX
Largest value representable in an object of type **unsigned char**; it must be at least 255.

UINT_MAX
Largest value representable in an object of type **unsigned int**; it must be at least 65,535.

ULONG_MAX
Largest value representable in an object of type **unsigned long int**; it must be at least 4,294,967,295.

USHRT_MAX
Largest value representable in an object of type **unsigned short int**; it must be at least 65,535.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 257

See Also

Environment, header, numerical limits

link — Definition

To *link* a program means to resolve external references among individual source files. External references may refer to data or code that reside in another translation unit.

Some function calls may be resolved by the inclusion of the code for that function from a library, which consists of implementation-defined or user-defined functions.

See Also

compile, Definitions, linkage

linkage — Definition

The term *linkage* refers to the matching of an identifier with its namesakes across blocks of code, and among files of source code, pretranslated object modules, and libraries.

Identifiers can have internal linkage, external linkage, or no linkage. An identifier with external linkage is known across multiple translation units. An identifier with internal linkage is known only within one translation unit. An identifier with no linkage has no permanent storage allocated for it and is local to a function or block.

The following describes each type of linkage in more detail:

External linkage
The following identifiers have external linkage:

- Any identifier for a function that either has no storage-class identifier or is marked with the storage-class identifier **extern**, but excluding ones marked with the storage-class identifier **static**.
- Any global identifier that either has no storage-class identifier or is marked **extern**.

Internal linkage
The following identifiers have internal linkage:

- Any identifier marked **static**.
- Any identifier for a function that has file scope and is marked **static**.

No linkage
The following identifiers have no linkage:

- An identifier for anything that is not an object or function; e.g., a structure member, a **union** member, an enumeration constant, a tag, or a label.

- Any identifier declared to be a function parameter.
- An identifier local to a block (i.e., an **auto** object), that does not have file scope and is not marked **extern**.

An identifier with internal linkage may be up to at least 31 characters long, and may use both upper- and lower-case characters. An identifier with external linkage, however, may have up to at least six characters, and is not required to use both upper- and lower-case characters. These limits are implementation defined.

An object marked **extern** will have the same linkage as any previous declaration of the same object within that translation unit. If there is no previous declaration, the object has external linkage.

If an object appears in the same source file with external and internal linkage declarations, behavior is undefined. This is called a *linkage conflict*. It may occur if an object is first declared **extern**, then later re-declared to be **static**.

Cross-references

Standard, §3.1.2.2
The C Programming Language, ed. 2, p. 228

See Also

identifiers, name space, scope

locale.h — Header

Localization functions and macros
#include <locale.h>

locale.h is a header that declares or defines all functions and macros used to manipulate a program's locale. The Standard describes the following items within this header:

Type	
lconv	Structure for numeric formatting
Manifest constants	
LC_ALL	All locale information
LC_COLLATE	Locale collation information
LC_CTYPE	Locale character-handling information
LC_MONETARY	Locale monetary information
LC_NUMERIC	Locale numeric information
LC_TIME	Locale time information
Functions	
localeconv	initialize **lconv** structure
setlocale	set/query locale

Cross-references

Standard, §4.4
The C Programming Language, ed. 2, pp

See Also

localization

localeconv() — Localization (libc)

Initialize **lconv** structure
#include <locale.h>
struct lconv *localeconv(void);

localeconv initializes the structure **lconv** and returns a pointer to it. **lconv** describes the formatting of numeric strings. For more information about this structure, see **lconv**.

The function **setlocale** establishes all or part of pre-defined locale as the current locale. A call to **setlocale** with the macros **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may alter a portion of **lconv**.

Cross-reference

Standard, §4.4.2.1

See Also

lconv, localization, locale.h, setlocale

localization — Overview

The Standard introduced the concept of *localization* to C programming.

The Problem

C was originally designed to implement the UNIX operating system. As such, its formatting functions assumed that the Latin alphabet would be used (that is, the only characters 'a' through 'z' and 'A' through 'Z'), assumed that no accented characters would be required, and also assumed that numeric strings would be formatted as they are in the United States. Since its invention, however, C has grown out of its original setting and its original country: it is now used internationally to write a wide range of application software.

The Standard recognizes that C internally is based on the English language. That is, C's keywords and library names reflect its origin in English, and will continue to do so. Localization, however, allows an application program to use the character set and formatting information that is specific to a given country in certain aspects of the language.

A locale can be selected when the program is run, so applications can be user-selectable. It may include things like monetary formatting, but preserve the underlying data: only the presentation differs. Locales provide a standard way for software developers to use locale-specific information without having to "reinvent the wheel" for each locale.

If an implementation of C supports various locales, then that locale information need not be gathered by programmers who write applications software. Rather than each software house writing support for European collating conventions or Japanese monetary formatting conventions, the support is provided once, by the implementor, and in a standard fashion.

Locale Functions

The Standard describes two functions that can be used to access information specific to a given locale.

setlocale can be used in either of two ways: to set the current locale, or to query the current locale settings. Either part or all of a locale's strings can be set or queried.

localeconv initializes an instance of the structure **lconv** and returns a pointer to it. This structure holds information that can be used to print numeric and monetary strings. For more information on this structure, see the entry for **lconv**.

The macros that begin with **LC_** are defined in the header **locale.h**, and represent the categories of locales (also known as *locale strings*). The following describes the areas of C that are affected by locales.

Characters
A national character set may include characters that lie outside of the Latin alphabet. Typically, these characters are not recognized as alphanumeric characters by functions like **isalpha**. To tell the translator to use the alternative character table for a given locale, use the call

```
setlocale(LC_CTYPE, locale);
```

The character-handling routines that are defined in the header **ctype.h** will use this locale information. This will also affect the functions that handle multibyte characters, as described below.

Collation
The sorting of strings that include national characters may present a problem. Normal collation functions depend upon the ASCII character order, and therefore do not know where additional, locale-specific characters go within the national character set. The Standard describes two functions, **strcoll** and **strxfrm**, that may collate strings which contain locale-specific characters. To set the locale information needed by these functions (so they know which national character order is used), use the call

```
setlocale(LC_COLL, locale);
```

strcoll and **strxfrm** will work in accordance with the current locale setting.

Date and Time
Most countries have an idiomatic way to express the current date and time. To set the locale information needed by the function **strftime**, use the call:

```
setlocale(LC_TIME, locale);
```

strftime can read the locale and format date and time strings accordingly.

Decimal Point
Different countries may use different characters to mark the decimal point. Occasionally, one character is used to mark the point in a numeric string and another to mark it in a string that describes money. The structure **lconv** contains the field **decimal_point**, which points to the character used to mark the decimal point in a numeric string.

To set the locale for functions that read or print the decimal point, use the call:

```
setlocale(LC_NUMERIC, locale);
```

All functions that perform string conversion, formatted output, or formatted input must interpret this information so these characters will be handled properly.

Money Each country has its own way to format monetary values. The character that represents the national currency varies from country to country, as does such aspects as whether the symbol goes before or after the numerals, how a negative value is rendered, what character is used to express a monetary decimal point (it may not be the same as the numeric decimal point), and how many digits are normally printed after the decimal point.

To set the locale information for money, use the call:

```
setlocale(LC_MONETARY, locale);
```

The structure **lconv**, which is initialized by the function **localeconv**, holds information needed to render monetary strings correctly.

Multibyte characters

Many countries, e.g., Japan and China, use systems of writing that use more characters than can be represented within one byte. Many operating systems and terminal devices, however, can receive only seven or eight bits at a time. To skirt this problem, the Standard describes two ways to encode such extensive sets of characters: with *wide characters* and *multibyte characters*.

A wide character is of type **wchar_t**. This type, in turn, is defined as being equivalent to the integral type that can describe all of the unique characters in the character set. This type is used mainly to store such characters in a device-independent manner.

A multibyte character, on the other hand, consists of two or more **char**s that together are understood by the terminal device as forming a non-alphabetic character or symbol. One wide character may map out to any number of multibyte characters, depending upon the number of systems of multibyte characters that are commonly in use.

The Standard describes five functions that manipulate wide characters and multibyte characters: **mblen**, **mbstowcs**, **mbtowc**, **wcstombs**, and **wctomb**. The actions of these functions are determined by the locale, as set by **setlocale**. To set a locale for the manipulation of multibyte characters, use the following call:

```
setlocale(LC_CTYPE, locale);
```

The Standard does not describe the mechanism by which tables of multibyte characters are made available to these functions.

Thousands

Large numbers can be broken up into groups of thousands to make them easier to read. The manner of grouping, including the number of items in each group and the character used to indicate the start of a new group, is locale specific.

The structure **lconv**, which is initialized by the function **localeconv**, contains the fields **thousands_sep**, **mon_grouping**, and **grouping**, which hold this information.

Default Locale

The only locale required of all conforming implementations is the **C** locale. This is the minimum set of locale strings needed to translate C source code. For a listing of what constitutes the **C** locale, see **lconv**.

When a C program begins, it behaves as if the call

```
setlocale(LC_ALL, "C");
```

had been issued.

Mechanism for Setting Locales

The Standard does not describe the mechanism by which **setlocale** makes locale information available to other functions, and by which the other functions use locale information. It is left to the implementation.

Cross-reference

Standard, §4.4

See Also

compliance, lconv, Library, locale.h

Notes

The Standard's section on compliance states that any program that uses locale-specific information does not conform strictly to the Standard. Therefore, a program that uses any locale other than the

C locale is not strictly conforming. A programmer should not count on being able to port such a program to any other implementation or execution environment.

localtime() — Time function (libc)

Convert calendar time to local time
#include <time.h>
struct tm *localtime(const time_t ****timeptr*);**

localtime takes the calendar time pointed to by *timeptr* and breaks it down into a structure of type **tm**. Unlike the related function **gmtime**, **localtime** preserves the local time of the system. This local time includes conversion to daylight savings time, if applicable. The daylight savings time flag indicates whether daylight savings time is now in effect, *not* whether it is in effect during some part of the year. Note, too, that the time zone is set by **localtime** every time the value returned by

```
      getenv("TIMEZONE")
```

changes. See the entry for **TIMEZONE** for more information on how **Let's C** handles time zone settings.

localtime returns a pointer to the structure **tm** that it creates. This structure is defined in the header **time.h**.

Example

The following example recreates the function **asctime**.

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

char *month[12] = {
      "January", "February" "March", "April",
      "May", "June", "July", "August",
      "September", "October", "November", "December"
};

char *weekday[7] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"
};

main()
{
      char buf[20];
      time_t tnum;
      struct tm *ts;
      int hour = 0;

      /* get time from system */
      time(&tnum);

      /* convert time to tm struct */
      ts=localtime(&tnum);

      if(ts->tm_hour==0)
            sprintf(buf,"12:%02d:%02d A.M.",
                  ts->tm_min, ts->tm_sec);
      else
            if(ts->tm_hour>=12) {
                  hour=ts->tm_hour-12;
                  if (hour==0)
                        hour=12;
                  sprintf(buf,"%02d:%02d:%02d P.M.",
                        hour, ts->tm_min,ts->tm_sec);

            } else
                  sprintf(buf,"%02d:%02d:%02d A.M.",
                        ts->tm_hour, ts->tm_min, ts->tm_sec);

      printf("\n%s %d %s 19%d %s\n",
            weekday[ts->tm_wday], ts->tm_mday,
            month[ts->tm_mon], ts->tm_year, buf);

      printf("Today is the %d day of 19%d\n",
            ts->tm_yday, ts->tm_year);

      if(ts->tm_isdst)
            printf("Daylight Saving Time is in effect.\n");
      else
            printf("Daylight Saving Time is not in effect.\n");

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.3.4
The C Programming Language, ed. 2, p. 256

See Also

asctime, ctime, date and time, gmtime, local time, strftime, TIMEZONE< tm

Notes

localtime returns a pointer to a statically allocated data area that is overwritten by each successive call.

log() — Mathematics (libm)

Compute natural logarithm
#include <math.h>
double log(double *z*);

log computes and returns the natural (base *e*) logarithm of its argument *z*. It is the inverse of the function **exp**.

Handing **log** an argument less than zero triggers a domain error. Handing it an argument equal to zero triggers a range error.

Cross-references

Standard, §4.5.4.4
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, ldexp, log10, mathematics, modf

log10() — Mathematics (libm)

Compute common logarithm
#include <math.h>
double log10(double *z***);**

log10 computes and returns the common (base 10) logarithm of its argument *z*.

Handing **log10** an argument less than zero triggers a domain error. Handing it an argument equal to zero triggers a range error.

Cross-references

Standard, §4.5.4.5
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, ldexp, log, mathematics, modf

long double — Type

A **long double** is a data type that represents at least a double-precision floating-point number. It is defined as being at least as large as a **double**. In some environments, extra precision can be gained by representing values with it.

Like all floating-point numbers, a **long double** consists of one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *mantissa*, or the number upon which the exponent works. The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased. The structure of a **long double** and the range of values that it can encode are set in the following macros, all of which are defined in the header **limits.h**:

LDBL_DIG
This holds the number of decimal digits of precision. This must be at least ten.

LDBL_EPSILON
Where *b* indicates the base of the exponent (default, two) and *p* indicates the precision (or number of base *b* digits in the mantissa), this macro holds the minimum positive floating-point number *x* such that 1.0 + *x* does not equal 1.0, *b*^1-*p*. This must be at least 1E-9.

LDBL_MAX
This holds the maximum representable floating-point number. It must be at least 1E+37.

LDBL_MAX_EXP
This is the maximum integer such that the base raised to its power minus one is a representable finite floating-point number. No value is given for this macro.

LDBL_MAX_10_EXP
This holds the maximum integer such that ten raised to its power is within the range of representable finite floating-point numbers. It must be at least +37.

LDBL_MANT_DIG
This gives the number of digits in the mantissa. No value is given for this macro.

LDBL_MIN
This gives the minimum value encodable within a **long double**. This must be at least 1E-37.

LDBL_MIN_EXP
This gives the minimum negative integer such that when the base is raised to that power minus one is a normalized floating-point number. No value is given for this macro.

LDBL_MIN_10_EXP
This gives the minimum negative integer such that ten raised to that power is within the range of normalized floating-point numbers.

A **long double** constant is represented by the suffix **l** or **L** on a floating-point constant.

For information about common floating-point formats, see **float**.

Cross-references

Standard, §2.2.4.2, §3.1.2.4, §3.1.3.1, §3.5.2
The C Programming Language, ed. 2, p. 196

See Also

double, float, types

long int — Type

A **long int** is a signed integral type. This type can be no closer to zero than an **int**.

A **long int** can encode any number between **LONG_MIN** and **LONG_MAX**. These are macros that are defined in the header **limits.h**. They are, respectively, -2,147,483,647 and 2,147,483,647.

The types **long**, **signed long**, and **signed long int** are synonyms for **long int**.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

int, short int, types

longjmp() — Non-local jumps (libc)

Execute a non-local jump
#include <setjmp.h>
void longjmp(jmp_buf *environment*, **int** *rval***);**

A call to **longjmp** restores the environment that the function **setjmp** had stored within the array **jmp_buf**. Execution then continues not at the point at which **longjmp** is called, but at the point at which **setjmp** was called.

environment is the environment that had been saved by an earlier call to **setjmp**. It is of type **jmp_buf**, which is defined in the header **setjmp.h**.

longjmp returns the value *rval* to the original call to **setjmp**, as if **setjmp** had just returned. **rval** must be a number other than zero; if it is zero, then **setjmp** will return one.

Cross-references

Standard, §4.6.2.1
The C Programming Language, ed. 2, p. 254

See Also

non-local jumps, setjmp

Notes

Many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **longjmp** and **setjmp** will result in the creation of mysterious and irreproducible bugs.

longjmp will work correctly "in the contexts of interrupts, signals and any of their associated functions.” Also, **longjmp**’s behavior is undefined if it is used from within a function called by signal received during the handling of a different signal.

Experience has shown that **longjmp** should not be used within an exception handler. The Standard does not guarantee that programs will work correctly when **longjmp** is used to exit interrupts and signals. Experience has shown that even if the **longjmp** terminates the signal handler and returns successfully to the context of the **setjmp**, the program can easily fail to complete the very next function call it attempts, usually because the signal interrupted an update of a non-atomic data structure. The Standard guarantees that the implementations of **setjmp**, **longjmp**, and **signal** will work together; it cannot make any promises about the interactions of these services with other library functions or with user code. *Caveat utilitor.*

lseek() — Extended function (libc)

Set read/write position
long lseek(short *fd*, **short** *how*, **long** *where***);**

lseek sets the file-position indicator for stream *fp*. this changes the point where the next read or write operation will occur.

fd is the file’s file descriptor, which is returned by **open**.

where and *how* describe the new seek position. *where* gives the number of bytes that you wish to move the seek position. It is measured from the beginning of the file if *how* is zero, from the current seek position if *how* is set to one, or from the end of the file if *how* is set to two.

A successful call to **lseek** returns the new seek position. For example,

```
position = lseek(filename, 100L, 0);
```

moves the seek position 100 bytes past the beginning of the file; whereas

```
position = lseek(filename, 0L, 1);
```

merely returns the current seek position, and does not change the seek position at all. **lseek** returns **-1L** if an error occurs, such as seeking to a negative position.

lseek differs from its cousin **fseek** in that **lseek** is an MS-DOS call and uses a file descriptor, whereas **fseek** is a library function and uses a **FILE** pointer.

See Also

extended miscellaneous, fseek, ftell

Notes

MS-DOS writes data at a physical address corresponding to the seek address. Thus, if you seek 10,000 bytes past the current end of file and write a string, the string will be written 10,000 bytes past the old end of file, and all the intervening data will then be made part of the file.

Some operating systems, such as MS-DOS, set the displacement from the file descriptor in bytes; others, such as VAX VMS, set the displacement in sectors. If you want your programs to be fully portable, you should avoid handing an absolute value to **lseek**.

lseek is not described in the ANSI Standard. A program that uses it does not comply strictly with the Standard, and may not be portable to other compilers or environments.

lvalue — Definition

An *lvalue* designates an object in storage. An lvalue can be of any type, complete or incomplete, other than type **void**.

A *modifiable lvalue* is any lvalue that is *not* of the following types:

- An array type.
- An incomplete type.
- Any type qualified by **const**.
- A structure or **union** with a member whose type is qualified by **const**, or with a member that is a structure or **union** with a member that is so qualified.

Only a modifiable lvalue is permitted on the left side of an assignment statement.

An lvalue normally is converted to the value that is stored in the designated object. When this occurs, it ceases to be an lvalue. For some lvalues, however, this does *not* occur, as follows:

- Any array type.
- When the lvalue is the operand of the operators **sizeof**, unary **&**, **--**, or **++**.
- When the lvalue is the left operand of the **.** operator.
- When the lvalue is the left operand of any assignment operator.

An lvalue with an array type is normally converted to a pointer to the same type. The value of the pointer is the address of the first member of the array. The exceptions to this operation are as follows:

- When it is the operand of the operators **sizeof** or unary **&**.
- When it is a string literal that initializes an array of **char**.
- When it is a string literal of wide characters that initializes an array of **wchar_t**.

In addition to the restrictions listed above, the following are also *not* lvalues, and hence cannot appear on the left side of an assignment statement:

- String literals.
- Character constants.
- Numeric constants.

Cross-references

Standard, §3.2.2.1
The C Programming Language, ed. 2, p. 197

See Also

conversions, function designator, rvalue

Notes

The term itself originally came from the phrase *left value*; in an expression like

```
object = value;
```

the element to the left of the '=' is the object whose value is modified. Because the Standard distinguishes between lvalues and modifiable lvalues, it prefers to define lvalue as being a contraction of the phrase *locator value*.