D

daemon — Definition

A **daemon**, in the context of C programming, is a process that is designed to perform a particular task or control a particular device without requiring the intervention of a human operator.

See Also

Definitions, process

date and time — Overview

#include <time.h>

The Standard describes nine functions that can be used to represent date and time, as follows:

Time conversion

asctime	Convert broken-down time to text
ctime	Convert calendar time to text
gmtime	Convert calendar time to Universal Coordinated Time
localtime	Convert calendar time to local time
strftime	Format locale-specific time

Time manipulation

clock	Get processor time used by the program
difftime	Calculate difference between two times
mktime	Convert broken-down time into calendar time
time	Get current calendar time

These functions use the following structures:

clock_t	System time
time_t	Calendar time
tm	Broken-down time

Let's C defines **time_t** as a 32-bit number that holds the number of seconds since January 1, 1970, 0h00m00s UTC.

The structure **tm** is defined as follows:

```
typedef struct tm {
      int tm_sec; /* second [0-60] */
      int tm_min; /* minute [0-59] */
      int tm_hour;/* hour [0-23]: 0 = midnight */
      int tm_mday;/* day of the month [1-31] */
      int tm_mon; /* month [0-11]: 0=January */
      int tm_year;/* year since 1900 A.D. */
      int tm_wday;/* day of week [0-6]: 0=Sunday */
      int tm_yday;/* day of the year [0-366] */
      int tm_isdst;/* daylight savings time flag */
} tm_t;
```

The member **tm_sec** can hold 61 seconds. This is done so that it can hold the "leap seconds" that are used internationally to help coordinate atomic clocks with pulsars and solar time.

Finally, the manifest constant **CLK_TCK** is used to convert the value returned by the function **clock** into seconds of real time. It is defined as being equivalent to one tick of the system clock. On the IBM PC and compatibles, this is equivalent to 18.206481933 milliseconds. This value does *not* change on machines that run at speeds higher than the standard 4.77 megahertz.

To print the local time, a program must perform the following tasks: First, read the system time with

time. Then, it must pass **time**'s output to **localtime**, which breaks it down into the **tm** structure. Next, it must pass **localtime**'s output to **asctime**, which transforms the **tm** structure into an ASCII string. Finally, it must pass the output of **asctime** to **printf**, which displays it on the standard output.

Let's C also includes numerous extensions to the ANSI Standard's time functions. These extensions increase the scope and accuracy of the Standard, and ease calculation of some time elements. See the entry on **extended time** for more information.

Cross-references

Standard, §4.12
The C Programming Language, ed. 2, pp. 255*ff*

See Also

broken-down time, calendar time, daylight saving time, extended time, local time, Library, TIMEZONE, universal coordinated time

dayspermonth() — Extended function (libc)

Return number of days in a given month
#include <xtime.h>
int dayspermonth(int *month***, int** *year***);**

dayspermonth returns the number of days in a given month of a given year A.D. *month* is the number of the month in question, from one to 12. *year* is the year A.D. in which *month* appears. Note that there is no year 0.

See Also

extended time, isleapyear, xtime.h

Notes

To conform to the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be rewritten.

DBL_DIG — Manifest constant

#include <float.h>
DBL_DIG is a manifest constant that is defined in the header **float.h**. It is an expression that defines the number of decimal digits of precision representable in an object of type **double**. It is defined to be ten.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 258

See Also

float.h, numerical limits

decimal-point character — Definition

The *decimal-point character* as being the character that marks the beginning of the fraction in a floating-point number. How this character is represented depends upon the program's locale. The locale specifier **LC_NUMERIC** describes how a particular locale represents the decimal-point character. In the **C** locale, it is the period '.'.

This character is used by the functions that convert a floating-point number to a string, or read a string and convert it to a floating-point number, i.e., **atof**, **fprintf**, **fscanf**, **printf**, **scanf**, **sprintf**, **sscanf**, **strtod**, **vfprintf**, **vprintf**, and **vsprintf**. This character is *not* used within C source; for

example,

```
sqrt(1,2);
```

passes two integer constants to **sqrt**, even if ',' is the decimal-point character for the current locale. Therefore, to print a C source file use the **C** locale, even if the program establishes another locale.

Cross-reference

Standard, §4.1.1

See Also

Definitions, localization

declarations — Overview

A *declaration* gives the type, storage class, linkage, and scope of a given identifier.

If a declaration also causes storage to be allocated for the object declared, then it is called a *definition*.

Declarators may be within a list, separated by commas. Each declarator has the type given at the beginning of the list, although a declarator may also have additional type information. For example,

```
int example1, *example2;
```

declares two variables: **example1** has type **int**, whereas **example2** has type "pointer to **int**."

Objects may be initialized when they are declared. See **initialization** for more information.

Cross-references

Standard, §3.5
The C Programming Language, ed. 2, pp. 210*ff*

See Also

bit-fields, declarators, definition, initialization, Language, linkage, scope, storage-class specifiers, type qualifiers, type specifiers

declarators — Overview

A *declarator* consists of an object being declared plus its array, pointer, and function modifiers.

For example,

```
int arrayname[10];
```

declares an array.

```
int functionname( int arg1, int arg2, char *arg3 );
```

declares a function.

```
int *pointername;
```

declares a pointer.

An implementation must be able to support at least 12 levels of declarators. Most implementations had given a lower limit.

Cross-references

Standard, §3.5.4
The C Programming Language, ed. 2, pp. 215*ff*

See Also

array declarators, declarations, function declarators, pointer declarators

Notes

To clarify some terminology that may be confusing:

A *declaration* encompasses the object declared, plus its specifiers, qualifiers, and levels of declarators.

A *declarator* consists of the object declared, plus its levels of specifiers (which set array dimensions, functions, or pointers).

A *definition* is a declaration that allocates storage.

default — C keyword

Default entry in switch table

default is a label that marks the default entry in the body of a **switch** statement. If none of the **case** labels match the value of the **switch** statement's conditional expression, then the **switch** statement jumps to the point marked by the **default** label, and begins execution from there.

Example

For an example of this label, see **printf**.

Cross-references

Standard, §3.6.1
The C Programming Language, ed. 2, p. 58

See Also

C keywords, case, statements, switch

Notes

A **switch** statement is not required to include a **default** label, but it is good programming practice to include one.

defined — C keyword

Check if identifier is defined
defined(*identifier* **)**
defined *identifier*

The Standard describes a new C keyword, **defined**. This keyword is used to check if *identifier* has been defined as macro or manifest constant. The preprocessing directives

```
#if defined(identifier)
```

and

```
#if defined identifier
```

have exactly the same effect as the directive:

```
#ifdef identifier
```

The **defined** operator is permitted only within **#if** and **#elif** expressions. It may not be used in any other context.

defined is not a reserved word. It can be used in more complex conditional statements, i.e.:

```
#if LEVEL==3 && defined FOO
```

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#if, #ifdef, keywords, preprocessing

definition — Definition

A *definition* is a declaration that also allocates storage for the item declared. For example,

```
int example[];
```

declares that **example** names an array of **int**s. Because the declaration does not say how large of an array **example** is, no memory is reserved; thus, this is a declaration but not a definition.

However, the declaration

```
int example[10];
```

declares that **example** names an array of ten **int**s. Because the declaration states how large **example** is, an appropriately sized portion of memory is reserved for it. Thus, this declaration is also a definition.

declaration and *definition* are easily confused, because the words are used in ways that are somewhat contrary to their normal English meanings.

A function definition is a special kind of definition that operates by its own rules. See **function definition** for more details.

Cross-references

Standard, §3.5
The C Programming Language, ed. 2, pp. 201, 210

See Also

declarations, function definition

Definitions — Overview

These definitions apply to topics throughout this Lexicon:

address
alias
alignment
argument
arena
ASCII
behavior
BIOS
bit
bit-fields
bit map
block
buffer
byte
compliance
cc0
cc1

cc2
cc3
cc4
daemon
decimal-point character
directory
domain error
executable file
false
field
file
file descriptor
interrupt
letter
link
manifest constant
nested comments
nybble
object format
object
parameter
pattern
port
portability
process
pun
quiet change
random access
range error
ranlib
read-only memory
record
register
rvalue
spirit of C
stack
Standard
standard error
standard input
standard output
stream
string
true
Universal Coordinated Time
wildcards

Cross-references

Standard, §1.6

See Also

LEXICON

diagnostics — Overview

The term *diagnostics* has two meanings in the ANSI Standard. The first is a set of macros that are used to test an expression at run time. The second refers to the way **Let's C** warns a user that a program contains an error.

Run-Time Diagnostics

The Standard describes a mechanism whereby an expression can be tested at run time. The macro **assert** tests the value of a given expression as the program runs. If the expression is false, **assert** prints a message into the standard error stream and then calls **abort**.

assert is defined in the header **assert.h**. This header also defines the manifest constant **NDEBUG**. If you define this macro before including **assert.h**, **assert** is redefined as follows:

```
#define assert(ignore)
```

This turns off **assert**. If an expression evaluated by **assert** has any side effects, using **NDEBUG** will change the program's behavior.

Diagnostic Warnings

Let's C produces a diagnostic for every translation unit that contains one or more errors of syntax rules or syntax constraints. A diagnostic can be either a fatal error, which prints a message and aborts translation, or simply a warning that prints a message and allows translation to proceed.

Cross-reference

Standard, §2.1.1.3

See Also

Library

difftime() — Time function (libc)

Calculate difference between two times
#include <time.h>
double difftime(time_t *newtime*, **time_t** *oldtime*);

difftime subtracts *oldtime* from *newtime*, and returns the difference in seconds.

Both arguments are of type **time_t**, which is defined in the header **time.h**.

Example

This example uses **difftime** to show an arbitrary time difference.

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

main(void)
{
      time_t      t1, t2;

      time(&t1);
      printf("Press enter when you feel like it.\n");
      getchar();
      time(&t2);

      printf("You waited %f seconds\n", difftime(t2, t1));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.2
The C Programming Language, ed. 2, p. 256

See Also

clock, date and time, mktime, time_t

digit — Definition

A *digit* is any of the following characters:

```
0  1  2  3  4  5  6  7  8  9
```

Cross-reference

Standard, §3.1.2

See Also

identifiers, nondigit

directory — Definition

A **directory** is a table that maps names to files. In other words, it associates the names of a file with their locations on the mass storage device. Under some operating systems, directories are also files, and can be handled like a file.

Directories allow files to be organized on a mass storage device in a rational manner, by function or owner.

See Also

Definitions, file, path

div() — General utility (libc)

Perform integer division
#include <stdlib.h>
div_t div(int *numerator*, **int** *denominator*);

div divides *numerator* by *denominator*. It returns a structure of the type **div_t**. This structure consists of two **int** members, one named **quot** and the other **rem**. **div** writes the quotient into **quot** and the remainder into **rem**.

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

For an example of this function, see **memchr**.

Cross-references

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

See Also

/, div_t, general utilities, ldiv

LEXICON

Notes

The Standard includes this function to permit a useful feature found in most versions of FORTRAN, where the sign of the remainder will be the same as the sign of the numerator. 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 **div** is undefined.

div_t — Type

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

div_t is a typedef that is declared in the header **stdlib.h**. It is the type returned by the function **div**.

div_t is a structure that consists of two **int** members, one named **quot** and the other **rem**. **div** writes its quotient into **quot** and its remainder into **rem**.

Example

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

Cross-references

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

See Also

div, general utilities, integer arithmetic, stdlib.h

do — C keyword

Loop construct
do { *statement* } **while(***condition***);**

do establishes conditional loop. Unlike the loops established by **for** and **while**, the condition in a **do** loop is evaluated *after* the operation is performed. This guarantees that at least one iteration of the loop will be executed.

do always works in tandem with **while**. For example

```
do {
      puts("Next entry? ");
      fflush(stdout);
} while(getchar() != EOF);
```

prints a prompt on the screen and waits for the user to reply. The **do** loop is convenient in this instance because the prompt must appear at least once on the screen before the user replies.

Cross-references

Standard, §3.6.5.2
The C Programming Language, ed. 2, p. 63

See Also

break, C keywords, continue, for, statements, while

dos.h — Header

Define MS-DOS functions and devices
#include <dos.h>

dos.h is the header that defines MS-DOS functions and devices. It is used with functions that directly interface with MS-DOS, such as **intcall**.

See Also

header, intcall, signals/interrupts

DOS-specific features — Overview

Let's C includes many features that relate specifically to the IBM PC, including the following:

- Source code
- Commands to be used with **Let's C**
- Example programs

This manual also includes a number of articles that given information about the i8086 and MS-DOS. See the Lexicon entry **technical information** for a list of these articles.

See Also

Lexicon, archive, command, example, technical information

double — C keyword

A **double** is a data type that represents a double-precision floating-point number. It is defined as being at least as large as a **float** and no larger than a **long double**.

Like all floating-point numbers, a **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 format of a **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**:

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

DBL_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.

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

DBL_MAX_EXP
: This is the maximum integer such that the base raised to its power minus one is a representable floating-point number.

DBL_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.

DBL_MANT_DIG
: This gives the number of digits in the mantissa.

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

DBL_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.

DBL_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. It must be at least -37.

For information on 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. 211

See Also

float, long double, types

dsreg() — i8086 support (libc)

Get value from DS segment register
#include <dos.h>
unsigned dsreg(void)

dsreg returns the value from from the i8086 DS register, which points to the base of the data segment.

Example

For an example of this function, see the entry for **csreg**.

See Also

csreg, esreg, i8086 support, ssreg

dup() — Extended function (libc)

Duplicate a file descriptor
short dup(short *fd***);**

dup duplicates the existing file descriptor *fd*, and returns the new descriptor. The returned value is the smallest file descriptor that is not already in use by the calling process. It returns a negative number when an error occurs, such as a bad file descriptor or no file descriptor available.

See Also

dup2, extended miscellaneous, fdopen, open

dup2() — Extended function (libc)

Duplicate a file descriptor
short dup2(short *fd***,** *newfd***);**

dup2 duplicates the file descriptor *fd*. Unlike its cousin **dup**, **dup2** allows you to specify a new file descriptor *newfd*, rather than have the system select one. If *newfd* is already open, the system closes it before assigning it to the new file. **dup2** returns the duplicate descriptor. It returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

See Also

dup, extended miscellaneous, fdopen, open