T

tag — Definition

A *tag* is a name that follows the keywords **struct**, **union**, or **enum**. It names the type of object so declared. For example, in the following code

```
	struct STR {
		. . .
	};
```

the identifier **STR** is a tag. It defines a new type of structure called **STR**. It does not, however, allocate any storage for any instance of this type.

Cross-references

Standard, §3.1.2.6
The C Programming Language, ed. 2, pp. 212*ff*

See Also

member, name space

tail — Command

Print the end of a file
tail [+*n***[bcl]] [***file***]**
tail [-*n***[bcl]] [***file***]**

tail copies the last part of *file*, or of the standard input if none is named, to the standard output.

The given *number* tells **tail** where to begin to copy the data. Numbers of the form +*number* count from the beginning of the file; those of the form -*number* count from the end of the file.

A specifier of blocks, characters, or lines (*b*, **c**, or **l**, respectively) may follow the number; the default is lines. If no *number* is specified, a default of -10 is assumed.

See Also

commands, egrep

Notes

Because **tail** buffers data measured from the end of the file, large counts may not work.

tan() — Mathematics (libm)

Calculate tangent
#include <math.h>
double tan(double *radian***);**

tan calculates and returns the tangent of its argument *radian*, which must be in radian measure.

Cross-references

Standard, §4.5.2.7
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, atan2, cos, mathematics, sin

tanh() — Mathematics (libm)

Calculate hyperbolic tangent
#include <math.h>
double tanh(double *value***);**

tanh calculates the hyperbolic tangent of *radian*.

Cross-references

Standard, §4.5.3.3
The C Programming Language, ed. 2, p. 251

See Also

cosh, mathematics, sinh

technical information — Overview

The Lexicon includes the following articles that give technical information on the IBM PC and MS-DOS:

ansi.sys	Device driver for console
BIOS data area	List magic areas within memory
byte ordering	Describe ordering of bytes
i8087	Floating-point co-processor
keyboard	Give keyboard scan codes
LARGE model	Describe Intel multi-segment memory model
model	Describe Intel memory models
SMALL model	Describe Intel single-segment memory model

See Also

DOS-specific information

tempnam() — Extended function (libc)

Generate a unique name for a temporary file
char *tempnam(char ****directory*, char ****name***);**

tempnam constructs a unique temporary name that can be used to name a file.

directory points to the name of the directory in which you want the temporary file written. If this variable is NULL, **tempnam** reads the environmental variable **TMPDIR** and uses it for *directory*. If neither *directory* nor **TMPDIR** is given, **tempnam** uses **\tmp**.

name points to the string of letters that will prefix the temporary name. This string should not be more than three or four characters, to prevent truncation or duplication of temporary file names. If *name* is NULL, **tempnam** will set it to **t**.

tempnam uses **malloc** to allocate a buffer for the temporary file name it returns. If all goes well, it returns a pointer to the temporary name it has written. Otherwise, it returns NULL if the allocation fails or if it cannot build a temporary file name successfully.

See Also

extended miscellaneous, mktemp, TMPDIR, tmpfile, tmpnam

Notes

tempnam is not described in the ANSI Standard. Programs that use it will not conform strictly the Standard, and may not be portable to other compilers or environments.

time() — Time function (libc)

Get current calendar time
#include <time.h>
time_t time(time_t **tp*);

The function **time** returns the current calendar time. If *tp* is set a value other than NULL, then **time** writes the result to the object pointed to by *tp*. **Let's C** defines the current system time as the number of seconds since January 1, 1970, 0h00m00s UTC.

time returns an object of the type **time_t**, which is defined in the header **time.h**. If the current calendar time is not available, **time** returns -1 cast to type **time_t**.

Example

This example displays the time.

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

main(void)
{
      time_t t;

      /* get the time */
      if(-1 == time(&t))
            printf("The time is unavailable?");
      else
            /* display it */
            printf(ctime(&t));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.4
The C Programming Language, ed. 2, p. 256

See Also

clock, date and time, difftime, mktime, time_t

time — Command

Print current time/Time execution of a command
time
time *command*
time " *command arguments* **"**

The command **time** performs two different tasks, depending upon whether it is used with or without arguments.

When **time** is typed without any arguments, it prints the date and time. The date and time are presented in a string of the form:

```
      Thu Apr  7 10:35:53 1988 CDT
```

The extension "CDT" stands for "Central Daylight Time". Daylight savings time will be returned only if the macro **TIMEZONE** is set properly in your **profile**. See **TIMEZONE** for more information.

If **time** is used with one or more arguments, it times the execution of a command. For example, typing **time ls** prints the contents of the current directory, then prints a string of the form:

```
00:00:02.340
```

which states how long the command took to execute.

If you wish to time a command that takes arguments, you must enclose the command and its arguments within quotation marks. For example, to time how long it takes to compile the program **window.c** with the **-VGEM** option to the compiler, use the command:

```
time "cc -VGEM window.c"
```

See Also

commands, date, msh, time (overview)

time.h — Header

Header for date and time
#include <time.h>

time.h is the header that declares the function and defines the types used to represent time. It contains prototypes for the following nine functions:

asctime	Convert broken-down time into text
clock	Get processor time used by the program
ctime	Convert calendar time to text
difftime	Calculate difference between two times
gmtime	Convert calendar time to Universal Coordinated Time
localtime	Convert calendar time to local time
mktime	Convert broken-down time into calendar time
strftime	Format locale-specific time
time	Get current calendar time

It also contains definitions for the following data types:

clock_t	Encode system time
time_t	Encode calendar time
tm	Encode broken-down time

It contains a definition for the macro **CLK_TCK**, which is used to convert the value returned by the function **clock** into seconds of real time.

Cross-references

Standard, §4.12
The C Programming Language, ed. 2, p. 255

See Also

CLK_TCK, date and time, header, xtime.h

time_t — Type

Calendar time
#include <time.h>

time_t is a data type that is defined in the header **time.h**. It is an arithmetic type that can represent time.

time_t is used to hold the calendar time, as computed from the system time by the function **time**. The functions **localtime** and **gmtime** use **time_t** to generate broken-down time, and the function **ctime** uses it to create a string that states the current date and time. The function **mktime** reads broken-down time and returns calendar time of type **time_t**.

Example

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

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

broken-down time, calendar time, clock_t, date and time

time_to_jday() — Extended function (libc)

Convert system time to Julian date
#include <time.h>
#include <xtime.h>
jday_t time_to_jday(time_t *time***);**

time_to_jday converts system time to Julian days. *time* is the current system time. It is declared to be of type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. **Let's C** defines the current system time as being the number of seconds from January 1, 1970, 0h00m00s UTC. The function **time** returns the current system time in this format.

time_to_jday returns the structure **jday_t**, which is defined in the header **xtime.h**. **jday_t** consists of two **unsigned long**s. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

See Also

extended time, jday_to_time, jday_to_tm, tm_to_jday, xtime.h

Notes

To conform with 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 altered.

TIMEZONE — Environmental variable

Time zone information
TIMEZONE=*standard***:***offset***[:***daylight***:** *date***:***date***:***hour***:***minutes***]**

TIMEZONE is an environmental parameter that holds information about the user's time zone. This information is used by **Let's C**'s time routines to construct their description of the current time and day.

To set **TIMEZONE**, use the **set** command, as follows:

```
        set TIMEZONE=description
```

where *description* is the string that describes your time zone. What this string consists of will be described below. Most users write this command into the file **AUTOEXEC.BAT**, so that **TIMEZONE** is set automatically whenever they reboot their system.

The Description String

A **TIMEZONE** description string consists of seven fields that are separated by colons. Fields 1 and 2 must be filled; fields 3 through 7 are optional.

Field 1 gives the name of your standard time zone. Field 2 gives the time zone's offset from Universal Coordinated Time (UTC) in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich. For example, users in Chicago set these fields as follows:

```
TIMEZONE=CST:360
```

CST is an abbreviation for Central Standard Time, that area's time zone; and 360 refers to the fact that Chicago's time zone is 360 minutes (six hours) behind that of Greenwich.

Field 3 gives the name of the local daylight saving time zone. In Chicago, for example, this field would be set as follows:

```
TIMEZONE=CST:360:CDT
```

CDT is an abbreviation for Central Daylight Time. The absence of this field indicates that your area does not use daylight saving time.

Fields 4 and 5 specify the dates on which daylight saving time begins and ends. If field 3 is set but fields 4 and 5 are not, changes between standard time and daylight saving time will be assumed to occur at the times legislated in the United States in 1986: at 2 A.M. standard time on the first Sunday in April, and at 2 A.M. daylight saving time on the last Sunday in October.

Fields 4 and 5 each consist of three numbers separated by periods. The first number specifies which occurrence of the day in the month marks the change, counting positive occurrences from the beginning of the month and negative occurrences from the the end of the month. The second number specifies a day of the week, numbering Sunday as one. The third number specifies a month of the year, numbering January as one. For example, in Chicago fields 4 and 5 are set to the following:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10
```

If the first number in either field is set to zero, then the last two numbers are assumed to indicate an absolute date. This is done because some countries switch to daylight saving time on the same day each year, instead of a given day of the week.

Finally, fields 6 and 7 specify the hour of the day at which daylight saving time begins and ends, and the number of minutes of adjustment. In Chicago, these are set as follows:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10:2:60
```

The '2' of field 6 indicates that the switch to daylight savings time occurs at 2 A.M. The "60" of field 7 indicates that daylight savings time changes the local time by 60 minutes. Although 60 minutes is the standard change, some regions of the world shift by 30, 45, 90, or 120 minutes; the last shift is also called "double daylight saving time".

For an example of this variable's use in a program, see the entry for **asctime**.

See Also

environmental variable, time

Notes

This environmental variable should be set only if you have set your computer system's time to conform with UTC. Otherwise, it will cause such functions as **localtime** to incorrectly offset the time they return.

For those requiring more information on this subject, see *Time Changes in the World*, compiled by Doris Chase Doane (three volumes, Hollywood, CA, Professional Astrologers, Inc., 1970).

tm — Type

Encode broken-down time
#include <time.h>

tm is the structure that holds the elements of broken-down time. It contains the following fields. (The values representable are shown within parentheses):

int tm_sec	Second (0-59)
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

The field **tm_isdst** indicates whether daylight saving time is currently in effect. It is set to a positive number if daylight saving time is in effect, to zero if it is not, and to a negative number if information concerning daylight saving time is not available.

The functions **localtime** and **gmtime** read the calendar time, as returned by the function **time**, and use it initialize **tm**; they then return a pointer to the structure.

The function **strftime** reads **tm** and uses it to build strings that present the date and time in a locale-specific manner. Finally, the function **mktime** reads **tm** and uses its contents to compute the corresponding calendar time.

Example

For an example of using this structure in a program, see **localtime**.

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

broken-down time, calendar time, clock_t, date and time, time_t

tm_to_jday() — Extended function (libc)

Convert calendar format to Julian time
#include <time.h>
#include <xtime.h>
jday_t tm_to_jday(tm **time***);**

tm_to_jday converts the system time, as described in the system calendar format, to Julian time.

time points to a copy of the structure **tm**, which is defined in the header **time.h**. The functions **gmtime** and **localtime** return the current time in this format. For more information on **tm**, see the entry for **time**.

tm_to_jday returns the structure **jday_t**, which is defined in the header **xtime.h**. **jday_t** to consist of two **unsigned long**s. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

See Also

extended time, jday_to_time, jday_to_tm, time, time_to_jday, xtime.h

Notes

To conform with 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 altered.

TMPDIR — Environmental variable

Directory that holds temporary files

TMPDIR names the directory into which **Let's C** writes its temporary files. If this variable is not set, the default is the directory in which the source files are kept. Note that this variable need be set only if space is a problem on the storage device that holds your current directory. For example, the command

```
set TMPDIR=a:\tmp
```

typed at the system prompt tells **cc** to write temporary files in the directory **tmp** on drive A:.

It is a good idea to set **TMPDIR** in **autoexec.bat**, to ensure that it is always set correctly.

See Also

cc, environmental variable

tmpfile() — STDIO (libc)

Create a temporary file
#include <stdio.h>
FILE *tmpfile(void);

tmpfile creates a file to hold data temporarily. The file is opened into binary update mode (**wb+**) and is removed automatically when it is closed or when the program ends. There is no way to access the temporary file by name. If your program needs to do so, it should open a file explicitly.

tmpfile returns NULL if it could not create the temporary file. If it could, it returns a pointer to the **FILE** associated with the temporary file. The function **exit** removes all files created by **tmpfile**.

Example

This example implements a primitive file editor that can edit large files. It uses two temporary files to keep all changes. The editor accepts the following commands:

dn	delete; **d52** deletes line 52
in	insert; **i7** inserts line before line 7
pn	print; **p17** prints line 17
p	print the entire file
w	write the edited file and quit
q	quit without writing the file

The entire temporary file is copied with each command.

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

FILE *fp, *tmp[2];
int linecount;
```

```
void
fatal(char *format, ...)
{
      va_list argptr;

      /* if there is a system message, display it */
      if(errno)
            perror(NULL);

      /* if there is a user message, use it */
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

/*
 * Copy up to line number or EOF.
 * Return number of lines copied.
 */
static int
copy(int line, FILE *ifp, FILE *ofp)
{
      int i, c, count;

      count = 0;
      for(c=i=1; (i<line || line==-1) && c!=EOF; i++) {
            while((c = fgetc(ifp)) != EOF && c != '\n')
                  fputc(c, ofp);

            if(c == '\n') {
                  count++;
                  fputc('\n', ofp);
            }
      }
      return(count);
}

/*
 * Read a file until line number is read.
 * Return 1 if line is found before EOF.
 */
static int
find(int line, FILE *ifp)
{
      int i, c;

      for(c=i=1; i<line && c!=EOF; i++)
            while((c = fgetc(ifp)) != EOF && c != '\n')
                  ;
      return(c != EOF);
}

main(int argc, char *argv[])
{
      int i, line, args;
      char c, cmdbuf[80];

      if(argc != 2)
            fatal("usage: tmpfile filename\n");
```

```
if((tmp[0]=tmpfile())==NULL||(tmp[1]=tmpfile())==NULL)
      fatal("Error opening tmpfile\n");

if((fp = fopen(argv[1], "r")) == NULL)
      fatal("Error opening %s\n", argv[1]);

linecount = copy(-1, fp, tmp[i = 0]);
fclose(fp);

/* one file pass per command */
for(;;) {
      if(gets(cmdbuf) == NULL)
            fatal("EOF on stdin\n");

      if(!(args = sscanf(cmdbuf, "%c%d", &c, &line)))
            continue;
      fseek(tmp[i], 0L, SEEK_SET);

      switch(c) {
      /* Write edited file */
      case 'w':
            if((fp = fopen(argv[1], "w")) ==  NULL)
                  fatal("Error opening %s\n", argv[1]);
            copy(linecount + 1, tmp[i], fp);
            fclose(fp);

      /* Quit */
      case 'q':
            exit(EXIT_SUCCESS);

      /* Print entire file */
      case 'p':
            if(args == 1) {
                  copy(linecount + 1, tmp[i], stdout);
                  continue;
            }
            if(find(line, tmp[i]))
                  copy(2, tmp[i], stdout);
            continue;

      /* Delete a line */
      case 'd':
            if(args == 1)
                  printf("dn where n is a number\n");
            else if(line > linecount)
                  printf("only %d lines\n", linecount);

            else {
                  copy(line, tmp[i], tmp[i^1]);
                  if(find(2, tmp[i]))
                        copy(-1, tmp[i], tmp[i^1]);

                  linecount--;
                  fseek(tmp[i], 0L, SEEK_SET);
                  i ^= 1;
            }
            continue;
            /* Insert a line */
            case 'i':
                  if(1 == args)
                        printf("in where n is a number\n");
                  else if(line > linecount)
                        printf("only %d lines\n", linecount);

                  else {
                        copy(line, tmp[i], tmp[i^1]);
                        printf("Enter inserted line\n");
                        copy(2, stdin, tmp[i^1]);
                        copy(-1, tmp[i], tmp[i^1]);
                        linecount++;

                        fseek(tmp[i], 0L, SEEK_SET);
                        i ^= 1;
                  }
                  continue;

            default:
                  printf("Invalid request\n");
                  continue;
            }
      }
}
```

Cross-references

Standard, §4.9.4.3
The C Programming Language, ed. 2, p. 243

See Also

mktemp, STDIO, tempnam, tmpnam

Notes

If a program exits abnormally or aborts, the files created by **tmpfile** may not be removed.

tmpnam() — STDIO (libc)

Generate a unique name for a temporary file
#include <stdio.h>
char *tmpnam(char ***name*);**

tmpnam constructs a unique name for a file. The names returned by **tmpnam** generally are mechanical concatenations of letters, and therefore are mostly used to name temporary files, which are never seen by the user. Unlike a file created by **tmpfile**, a file named by **tmpnam** does not automatically disappear when the program exits. It must be explicitly removed before the program ends if you want it to disappear.

name points to the buffer into which **tmpnam** writes the name it generates. If *name* is set to NULL, **tmpnam** writes the name into an internal buffer that may be overwritten each time you call this function.

tmpnam returns a pointer to the temporary name. Unlike the related function **tempnam**, **tmpnam** assumes that the temporary file will be written into directory **\tmp** and builds the name accordingly.

Example

The following example uses **tmpnam** to generate some file names, opens one, and writes the rest of the names into it.

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

void fatal(const char *string)
{
      fprintf(stderr, "%s\n", string);
      exit(EXIT_FAILURE);
}

main()
{
      int i, files;
      FILE *fp;
      char buffer[L_tmpnam];

      if ((fp = fopen(tmpnam(buffer), "w")) == NULL)
            fatal("Cannot open temporary file");
      printf("Temporary file name is %s\n", buffer);

      /* put realistic limit on number of names */
      100 > TMP_MAX ? files = TMP_MAX : files = 100;
      for(i = 0; i < files; i++)
            fprintf(fp, "%s\n", tmpnam(NULL));

      fclose(fp);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.4.4
The C Programming Language, ed. 2, p. 243

See Also

L_tmpnam, mktemp, STDIO, tempnam, tmpfile, TMP_MAX

Notes

If you want the file name to be written into *buffer*, you should allocate at least **L_tmpnam** bytes of memory for it; **L_tmpnam** is defined in the header **stdio.h**.

tmpnam can be called at least **TMP_MAX** times to return unique file names. **TMP_MAX** is also set in **stdio.h**.

toascii() — Extended macro (xctype.h)

Convert characters to ASCII
#include <xctype.h>
int toascii(int *c*);

toascii takes any integer value *c*, keeps the low seven bits unchanged, and changes the others to zero. This, in effect, transforms the integer value to an ASCII character. **toascii** then returns the transformed integer. If *c* is a valid ASCII character, it is returned unchanged.

Example

This example prompts for a file name. It then opens the file and prints its contents, while converting all non-alphanumeric characters to alphanumeric.

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

```
main(void)
{
      FILE *fp;
      int ch;
      int filename[20];

      printf("Enter file name: ");
      fflush(stdout);
      gets(filename);

      if ((fp = fopen(filename, "r")) != NULL) {
            while ((ch = fgetc(fp) != EOF)
                  putchar(isascii(ch) ? ch : toascii(ch));
      } else {
            printf("Cannot open %s\n", filename);
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

See Also

extended character handling

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **ctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

token — Definition

A *token* is the basic, indivisible unit of text that is processed by the translator.

There are two varieties of token: *lexical token* and *preprocessing token*. When the Standard uses the term "token," it refers to what is here called a "lexical token." Note, too, that the term "preprocessing token" does not mean a token that is manipulated only by the preprocessor.

Preprocessing tokens form the following varieties of lexical elements:

- Character constant.
- Header name.
- Identifier.
- Operator.
- Preprocessing number.
- Punctuator.
- String literal.
- Each non-white space character that does not fall into one of the above categories.

White-space characters can appear only within a header name, a character constant, or a string literal; in all other instances, white space separates tokens.

Preprocessing tokens are processed during phases 3 through 6 of translation. For details on translation, see the entry for **translation phases**. In brief, all preprocessing directives are executed: **#include** states are expanded, code is conditionally included, and macros are expanded. Each comment is replaced with one white-space character.

Adjacent string literals are concatenated and clusters of text that are not separated by white space are parsed. A cluster of text is always parsed into the longest possible sequence of characters that forms a valid token. For example, the text

```
        a+++++b
```

must be parsed into:

```
        a ++ ++ + b
```

The preprocessor passes unchanged what it does not recognize as being a preprocessor token.

Lexical tokens (which the Standard calls simply "tokens") form the following types of lexical elements:

- Constant.
- Identifier.
- Keyword.
- Operator.
- Punctuator.
- String literal.

Lexical tokens are parsed, analyzed, and linked.

Cross-references

Standard, §3.1
The C Programming Language, ed. 2, pp. 191, 229

See Also

lexical elements, translation phase

tolower() — Character handling (ctype.h)

Convert character to lower case
int tolower(int *c*);

The macro **tolower** converts the upper-case character *c* to its corresponding lower-case character, as defined by the locale's character set. The Standard defines an upper-case character as one for which the function **isupper** returns true. *c* must be a value that is representable as an **unsigned char** or **EOF**.

If *c* is an upper-case letter, then **tolower** returns the corresponding lower-case letter. If *c* is not a letter or is already lower case, then **tolower** returns it unchanged.

Example

The following example demonstrates **tolower** and **toupper**. It reverses the case of every character in a text file.

```
void fatal(const char *message)
{
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
}
```

```
#include <ctype.h>
#include <stdio.h>
void fatal(const char *string);

main(int argc, char *argv[])
{
      FILE *fp;
      int ch;

      if (argc != 2)
            fatal("usage: example filename");

      if ((fp = fopen(argv[1], "r")) == NULL)
            fatal("cannot open text file");

      while ((ch = fgetc(fp)) != EOF)
            putchar(isupper(ch) ? tolower(ch) : toupper(ch));
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.3.2.1
The C Programming Language, ed. 2, p. 249

See Also

character handling, toupper

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

toupper() — Character handling (libc)

Convert character to upper case
int toupper(int *c*);

toupper converts the lower-case character *c* to its corresponding upper-case character. The Standard defines an lower-case character as one for which the function **islower** returns true. *c* must be either a value that is representable as an **unsigned char** or **EOF**.

If **c** is an lower-case letter, then **toupper** returns the corresponding upper-case letter for the locale's character set. If *c* is not a letter or is already upper case, then **toupper** returns it unchanged.

Example

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

Cross-references

Standard, §4.3.2.2
The C Programming Language, ed. 2, p. 249

See Also

_toupper, character handling, tolower

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

translation unit — Definition

A *translation unit* is the basic unit of code that is translated into executable form. It consists of a source file, plus all headers that are requested with the preprocessing directive **#include**, and excluding all code that is skipped by preprocessing conditional inclusion.

Cross-references

Standard, §2.1.1.1
The C Programming Language, ed. 2, p. 191

See Also

#include, conditional inclusion, Environment, source file

trigraph sequences — Definition

A *trigraph sequence* is a set of three characters that represents one character in the C character set. The set of trigraph sequences was created to allow users to use the full range of C characters, even if their keyboards do not implement the full C character set. Trigraph sequences are also useful with input devices that reserve one or more members of the C character set for internal use.

Each trigraph sequence is introduced by two question marks. The third character in the sequence indicates which character is being represented. The following table gives the set of trigraph sequences:

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

The characters represented are the ones used in the C character set but not included in the ISO 646 character set. ISO 646 describes an invariant sub-set of the ASCII character set.

Trigraph sequences are interpreted even if they occur within a string literal or a character constant. This is because they are interpreted before the source code is tokenized; see **translation phases** for more information. Thus, strings that uses a literal "??" will not work the same as under a non-ANSI implementation of C. For example, the function call

```
printf("Feel lucky, punk??!\n");
```

would print:

```
Feel lucky, punk|
```

This is a silent change that may break existing code.

To print a pair of questions marks, use the escape sequence '\?\?'. For example:

```
printf("Feel lucky, punk\?\?!\n");
```

Cross-references

Standard, §2.2.1.1
The C Programming Language, ed. 2, p. 229

See Also

Environment

true — Definition

In the context of a C program, an expression is *true* if it yields nonzero.

See Also

Definitions, false

typedef — C keyword

Synonym for another type

The storage-class specifier **typedef** names a synonym for a type.

The new synonym must include all qualifiers and storage-class specifiers. For example, the declaration

```
typedef volatile unsigned long int giant;
```

states that the type **giant** is a synonym for **volatile unsigned long int**. Thus, the declaration

```
giant example();
```

declares, in effect, that the function **example** returns an **volatile unsigned long int**. An object declared to be type **giant** and one declared to be type **volatile unsigned long int** behave exactly the same.

typedef is often used to declare a structure type. For example, the structure declaration

```
typedef struct {
      int member1, member2;
      long member3;
} EXAMPLE;
```

declares that **EXAMPLE** is a type name, and that it is a synonym for the structure that precedes it.

Cross-references

Standard, §3.5.6
The C Programming Language, ed. 2, p. 146

See Also

storage-class specifiers, types

Notes

The term *typedef* also describes a type that is defined in a **typedef** statement.

The Standard does not allow benign redeclarations of typedefs. For example, if the declaration

```
typedef int SINT;
```

were included in a header and the same declaration appeared in a source file that included this header, a diagnostic message should appear during translation.

type qualifier — Overview

A *type qualifier* is, as its name implies, a keyword that alters the nature of a type in a significant way.

There are two type qualifiers:

const	Qualify an identifier as not modifiable
volatile	Qualify an identifier as changing frequently

The changes affected by a type qualifier take effect only in expressions that yield an lvalue.

No type qualifier may modify an identifier more than once, either directly or via a **typedef**. Also, two types are considered to be compatible only if their qualifiers match.

Many quirks surround the use of qualifiers. For example:

```
const int *cip;
int *ip;

cip = ip;        /* RIGHT */
ip = cip;        /* WRONG */
```

In effect, assignments that serve to "hide" the qualified object must be diagnosed. Although the above examples uses the qualifier **const**, the same restrictions apply to any combination of qualifiers on an object.

Cross-references

Standard, §3.5.3
The C Programming Language, ed. 2, p. 211

See Also

declarations

Notes

Because type qualifiers can alter the manner in which an object is accessed, they can be considered to be "access modifiers".

types — Overview

Type determines the meaning of a value stored in an object or returned by a function. For example, if an object four bytes long were declared to be type **long**, the meaning of its contents is quite different than if it were declared to be of type **long** *, or a pointer to a **long**. In the former instance, the contents are regarded as an absolute value. In the latter, the contents are regarded as an address of another object.

The Standard organizes types into a number of varieties and categories, as follows:

Aggregate types
All array and structure types.

Arithmetic types
The set of integral and floating types.

Array types
A set of objects that have the same type and are in contiguous memory.

Basic types
The set of **char**, the signed and unsigned integer types, and the floating types; i.e., arithmetic types but not enumerated types.

Composite type
A type constructed from two compatible types, one of which has additional type information. For example, the declarations

```
int example;
   . . .
```

```
      const int example;
```

together form a composite type.

Derived declarator types
The set of array, function, and pointer types.

Derived types
The set of array, function, pointer, structure, and **union** types that are derived from the basic types.

Enumerated type
A set of named integer constant values that comprise an enumeration.

Floating types
The types **float**, **double**, or **long double**.

Function types
The type that describes a given function with a specified return type and specified number and types of parameters.

Incomplete types
A type for which the translator does not possess all necessary information. Examples are an array of unknown size, or a structure or **union** of unknown content. An incomplete type must be completed by the end of translation.

Integral types
The set of type **char**, the signed and unsigned integer types, and the enumerated types.

Object types
The set of types that describe objects, rather than functions.

Pointer type
A type that describes the type of object to which a pointer points. The two classes of pointers are object pointers and function pointers. Object pointers are referred to by the type of object to which they point.

Qualified type
A type whose top type is qualified with some combination of the type qualifiers **const**, **noalias**, or **volatile**.

Scalar types
The set of arithmetic types and pointer types.

Signed integer types
Any of the types **signed char**, **int**, **long int**, or **short int**. Any of the last three types may also use the prefix **signed**, but the addition of this prefix does not change them in any way.

Structure type
A type that describes a group of data objects that are contiguous; each object may have its own specified type and its own name.

Top type
The top type of a basic type is the type itself. The top type of a derived type is the first type used to describe the type; for example, the type **int *** is described as "pointer to **int**"; therefore, its top type is pointer.

union *type*
A type that describes a set of objects that overlap in memory. Each object may have its own type and its own name.

Unqualified type
Any type whose top type is *not* qualified with the type qualifiers **const**, **noalias**, or **volatile**.

Unsigned integer types
Any of the types **unsigned char**, **unsigned int**, **unsigned long int**, and **unsigned short int**.

Basic Types

The following is the set of basic types. Those on the same line are synonyms:

char
double
float
int, signed int
long double
long int, long, signed long, signed long int
signed char
short int, short, signed short int, signed short
unsigned char
unsigned int
unsigned long int, unsigned long
unsigned short int, unsigned short

Data Formats

Mark Williams Company has written C compilers for a number of different computers. Each has a unique architecture and defines data formats in its own way.

The following table gives the sizes, in **char**s, of the data types as they are defined by various microprocessors.

Type	*i8086 SMALL*	*i8086 LARGE*	*Z8001*	*Z8002*	*68000*	*PDP11*	*VAX*
char	1	1	1	1	1	1	1
double	8	8	8	8	8	8	8
float	4	4	4	4	4	4	4
int	2	2	2	2	2	2	4
long	4	4	4	4	4	4	4
long double	8	8	8	8	8	8	8
pointer	2	4	4	2	4	2	4
short	2	2	2	2	2	2	2

Let's C places some alignment restrictions on data, which conform to all restrictions set by the microprocessor. Byte ordering is set by the microprocessor. See the Lexicon entry on **byte ordering** for more information.

Type Checking

C is not strongly typed, which means that it allows different types to be mixed relatively freely, and be changed (or **cast**) from one type to another.

Let's C checks types more strictly than the C standard implies. **Let's C**'s type checking can be enabled or disabled in degrees, using **-VSTRICT** and other "variant" options with the **cc** command.

Type Promotion

In arithmetic expressions, **Let's C** promotes one signed type to another signed type by sign extension, and promotes one unsigned type to another unsigned type by zero padding. For example, **char** promotes to **int** by sign extension, while **unsigned char** promotes to **unsigned int** by zero padding.

Cross-references

Standard, §3.1.2.5
The C Programming Language, ed. 2, p. 195

See Also

identifiers, signed, struct, type specifiers, union, unsigned

Notes

On some machines, **char** is a synonym for **signed char**. On others, it is a synonym for **unsigned char**. You should declare a **char** variable to be **signed** or **unsigned** if its behavior when promoted to **int** is significant.

type specifier — Overview

A *type specifier* specifies the type of an object or function when it is declared.

The following lists the legal C type specifiers:

char
double
enum *tag-name*
float
int
long
signed
short
struct *tag-name*
unsigned
union *tag-name*
void

The type specifiers can be combined into any one of the following combinations. Those on the same line are synonyms:

char
double
enum *type-name*
float
int, **signed**, **signed int**
long double
long int, **long**, **signed long**, **signed long int**
signed char
short int, **short, signed short int**, **signed short**
struct *type-name*
typedef *name*
union specifier
unsigned char
unsigned int, **unsigned**
unsigned long int, **unsigned long**
unsigned short int, **unsigned short**
void

Cross-references

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

See Also

types, enum, struct, typedef, union, void