U

ungetc() — STDIO (libc)

Push a character back into the input stream
#include <stdio.h>
int ungetc(int *character*, **FILE** **fp*);

ungetc converts *character* to an **unsigned char** and pushes it back into the stream pointed to by *fp*, where the next call to an input function will read it as the next character available from the stream. **ungetc** clears the end-of-file indicator for the stream.

The Standard only guarantees that one character can safely be pushed back into *fp* at any given time. A subsequent call to **fflush**, **fseek**, **fsetpos**, or **rewind** will discard the "ungotten" character.

ungetc returns *character* if it could be pushed back onto *fp*. Otherwise, it returns **EOF**. If *character* is equivalent to **EOF**, **ungetc** will fail.

Example

The following example opens a file and returns how many lines and sentences it contains. A sentence is defined as being any passage of text that ends in a period, a question mark, or an exclamation point.

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

void
fatal(char *message)
{
      fprintf(stderr, "%s0, message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *fp;
      int ch, nlines, nsents;
      nlines = nsents = 0;

      /* Check number of arguments */
      if (argc != 2)
            fatal("Usage: example filename");

      /* Open file to be read */
      if ((fp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open file for reading");

      else {
            /* read lines of text */
            while ((ch = fgetc(fp)) != EOF) {
                  /* increment line count */
                  if (ch == '\n') ++nlines;
```

```
                else if (ch == '.' || ch == '!' || ch == '?') {
                      /* check if period is an ellipsis */
                      if ((ch = fgetc(fp)) != '.') {
                            /* if not, bump sentence count */
                            ++nsents;
                            /* return extra char to stream */
                            ungetc(ch, fp);
                      }

                      /* skip ellipsis */
                      else for(ch='.'; (ch=fgetc(fp))=='.';)
                            ;
                }
          }

          printf("%d line(s), %d sentence(s).\n", nlines, nsents);
     }
     return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.11
The C Programming Language, ed. 2, p. 247

See Also

fgetc, getc, getchar, scanf, STDIO

Notes

How **ungetc** affects the file-position indicator will vary, depending upon whether *fp* was opened into text mode or binary mode. If *fp* was opened into binary mode, then its file-position indicator is decremented with every successful call to **ungetc**. If, however, it was opened into text mode, then the value of the file-position indicator after a successful call to **ungetc** is unspecified; the Standard specifies only that when a character is pushed back and then re-read, the file position indicator has same value as it did when the character was first read.

union — Type

A **union** is a data type whose members occupy the same region of storage. It is used when one value may be used in a number of different circumstances. This is in contrast with a **struct**, which is a set of data elements that are laid adjacent to each other. Each object within a **union** may have its own name and distinct type.

Any object type may be contained within a **union**, including a bit-field. No incomplete object may be used. Thus, a **union** may not contain a copy of itself, but it may contain a pointer to itself. A **union** is regarded is incomplete until its closing '}' is read.

The size of a **union** is that of its largest member. Thus, a pointer to a **union** can, if correctly cast, be used as a pointer to each of the **union**'s members.

In effect, a **union** is a multiple declaration of a variable. For example, a **union** may be declared to consist of an **int**, a **double**, and a **char** *. Any one of these three elements can be held by the **union** at a time, and will be handled appropriately by it. For example, the declaration

```
union {
      int number;
      double bignumber;
      char *stringptr;
} EXAMPLE;
```

allows **EXAMPLE** to hold either an **int**, a **double**, or a pointer to a **char**, whichever is needed at the time. The elements of a **union** are accessed like those of a **struct**: for example, to access **number** from the above example, type **EXAMPLE.number**.

unions are helpful in dealing with heterogeneous data, especially within structures. However, you must keep track of what data type the **union** is holding at any given time. Assigning to a **double** within a **union** and then reading the **union** as though it held an **int** will yield results that are defined by the implementation.

A **union** initializer may only initialize the *first* member of the **union**.

Example

The following example uses a **union** to demonstrate the byte ordering of the machine upon which the program is run. It assumes that an **int** is two bytes long, and a **long** is four bytes long.

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

main(void)
{
      union {
            char bytes[4];
            int words[2];
            long longs;
      } u;
      u.l = 0x12345678L;

      printf("%x %x %x %x\n",
            u.bytes[0], u.bytes[1], u.bytes[2], u.bytes[3]);
      printf("%x %x\n", u.words[0], u.words[1]);
      printf("%lx\n", u.longs);
      return EXIT_SUCCESS;
}
```

Cross-references

Standard, §3.1.2.5, §3.5.2.1
The C Programming Language, ed. 2, pp. 212*ff*

See Also

bit-field, member name, struct, tag, types

Notes

Oftentimes, **union** will be a member of a structure, and the preceding structure member will be a "tag" field, whose value indicates the type of object the **union** currently has stored. Though such a tag is required in some languages (such as Pascal), it is not required in C.

universal coordinated time — Definition

Universal coordinated time (*universel temps coordonne*, or UTC) is a universal standard of time that is based on study of an atomic clock, as corrected by comparison with pulsars. It is, for all practical purposes, identical to Greenwich Mean Time, which is the mean solar time recorded at the Greenwich Observatory in England, where by international convention the Earth's zero meridian is fixed.

Standard local time is usually calculated as an offset of UTC. For example, the time zone for Chicago is six hours (360 minutes) behind UTC, so the standard time for Chicago is calculated by subtracting 360 minutes from UTC. Calculating local time may not always be so easy, however. For example, some Islamic countries calculate local time by dividing the time between sunrise and sunset into 12 hours.

LEXICON

The function **gmtime** returns a pointer to the structure **tm** that has been initialized to hold the current UTC. The name of this function reflects the older practice of referring to Greenwich Mean Time instead of UTC.

Cross-reference

Standard, §4.12.1

See Also

broken-down time, calendar time, date and time, gmtime, local time, localtime

unlink() — Extended function (libc)

Remove a file
short unlink(char *_name_**);**

unlink removes the directory entry for the given file *name*, which in effect erases *name* from the disk. *name* cannot be open when **unlink**ed. The name is an historical artifact.

unlink returns -1 if it cannot remove a file, and zero if it can.

Example

This example removes the files named on the command line.

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

/* prototype for extended function */
extern short unlink(char *name);

main(int argc, char *argv[])
{
      int i;

      for (i = 1; i < argc; i++)
      {
            if (unlink(argv[i]) == -1)
            {
                  printf("Cannot unlink \"%s\"\n", argv[i]);
                  exit(EXIT_FAILURE);
            }
      }
      exit(EXIT_SUCCESS);
}
```

See Also

extended miscellaneous, remove

Notes

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

The ANSI function **remove** also removes files. It is recommended that you use it instead of **unlink** so that your programs will conform more strictly to the Standard.

unsigned — C keyword

When a declaration includes the modifier **unsigned**, it indicates that the type can hold only a non-negative value.

There are four **unsigned** data types: **unsigned char**, **unsigned int**, **unsigned long int**, and **unsigned short int**. If the modifier **unsigned** is not used, the translator assumes that **int**, **long int**, and **short int** are signed. The implementation defines whether **char** is signed or unsigned by default.

An unsigned data type takes the same amount of storage as the corresponding signed type, and has the same alignment requirements.

Any value that can be represented by both a signed and an unsigned type will be represented the same way in both. An unsigned type, however, cannot represent a negative value. The sign bit is freed to hold a value. In this instance, an unsigned type can store a value of twice what can be stored in its signed counterpart.

Arithmetic that involves unsigned types will never overflow. If an arithmetic operation produces a value that is too large to fit into a particular unsigned type, that value is divided by one plus the largest value that can be held in that unsigned type, and the remainder is then stored in the unsigned type.

For information about converting one type of integer to another, see **integral types**.

When **unsigned** is used by itself, it is regarded as a synonym for **unsigned int**.

Cross-references

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

See Also

char, signed, types, unsigned

unsigned char — Type

An **unsigned char** is an unsigned integral type. It takes the same amount of storage as a **char**, and has the same alignment requirements.

An **unsigned char** has the minimum value of zero, and a maximum value of **UCHAR_MAX**. The last is a macro that is defined in the header **limits.h**. It is 255.

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. 44

See Also

char, signed char, types, unsigned

unsigned int — Type

An **unsigned int** is an unsigned integral type. It requires the same amount of storage as a **int** and has the same alignment requirements.

An **unsigned int** has the minimum value of zero, and a maximum value of **UINT_MAX**. The last is a macro that is defined in the header **limits.h**. It is 65,535.

The type **unsigned** is a synonym for **unsigned 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, types, unsigned

unsigned long int — Type

An **unsigned long int** is an unsigned integral type. It requires the same amount of storage as a **long int**, and has the same alignment requirements.

An **unsigned long int** has the minimum value of zero, and a maximum value **ULONG_MAX**. The last is a macro that is defined in the header **limits.h**. It is 4,294,967,295.

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

long int, types, unsigned

unsigned short int — Type

An **unsigned short int** is an unsigned integral type. It requires the same amount of storage as a **short int**, and has the same alignment requirements.

An **unsigned short int** has the minimum value of zero, and a maximum value of **USHRT_MAX**. The last is a macro that is defined in the header **limits.h**. It is 65,535.

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

short int, types, unsigned