R

raise() — Signal handling (libc)

Send a signal
#include <signal.h>
int raise(int *signal***);**

raise sends *signal* to the program that is currently being executed. If called from within a signal handler, the processing of this signal may be deferred until the signal handler exits.

Example

This example sets a signal, raises it itself, then allows the signal to be raised interactivly. Finally, it clears the signal and exits.

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

void gotcha(void);

void
setgotcha(void)
{
      if(signal(SIGINT, gotcha) == SIG_ERR) {
            printf("Couldn't set signal\n");
            abort();
      }
}

void
gotcha(void)
{
      char buf[10];

      printf("Do you want to quit this program? <y/n> ");
      fflush(stdout);
      gets(buf);

      if(tolower(buf[0]) == 'y')
            abort();

      setgotcha();
}

main(void)
{
      char buf[80];

      setgotcha();
      printf("Set signal; let's pretend we get one.\n");
      raise(SIGINT);

      printf("Returned from signal\n");
      /* <ctrl-c> may not work on all operating systems */
      printf("Try typing <ctrl-c> to signal <enter> to exit");
      fflush(stdout);
      gets(buf);
```

```
      if(signal(SIGINT, SIG_DFL) == SIG_ERR) {
            printf("Couldn't lower signal\n");
            abort();
      }

      printf("Signal lowered\n");
      exit(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.7.2.1
The C Programming Language, ed. 2, p. 255

See Also

signal, signal handling, signal.h

Notes

This function is derived from the UNIX function **kill**.

rand() — General utility (libc)

Generate pseudo-random numbers
#include <stdlib.h>
int rand(void)

rand generates and returns a pseudo-random number. The number generated is in the range of zero to **RAND_MAX**, which equals 32,767.

rand will always return the same series of random numbers unless you change its *seed*, or beginning-point, with **srand**. Without having first called **srand**, it is as if you had initially set *seed* to one.

Example

This example produces a **char** that consists of random bits. The Standard's description of **rand** produces random **int**s, not random bits.

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

unsigned char
bitrand(void)
{
      register int i, r;

      for(i = r = 0; i < CHAR_BIT; i++) {
            r <<= 1;
            if(((long)rand() << 1) < (long)RAND_MAX)
                  r++;
      }
      return(r);
}

main(void)
{
      printf("Random stuff %02x %02x %02x\n",
            bitrand(), bitrand(), bitrand());
      return(EXIT_SUCCESS);
}
```

LEXICON

Cross-references

Standard, §4.10.2.2
The C Programming Language, ed. 2, p. 252

See Also

general utilities, RAND_MAX, srand

random access — Definition

In the context of computing, **random access** means that an entity can be accessed at any point, not just at the beginning. This means that all points within memory can be accessed equally quickly. This contrasts with *sequential access*, in which entities must be accessed in a particular order, so that some entities take longer to access than do others.

A tape drive is an example of a sequential access device, i.e., the order in which data are read is dictated by the order in which they stream past the tape head. Random-access memory (RAM) is an example of random access. Hard disks and floppy disks combine elements of random access and sequential access.

RAM, which usually consists of semiconductor integrated circuits, is also strictly random access. In this regard, the term "RAM" is slightly misleading; a more accurate name would be "read/write memory", to contrast RAM with read-only memory (ROM), which is also *random access* memory.

See Also

Definitions, read-only memory

read() — Extended function (libc)

Read from a file
short read(short *fd***, char** ***buffer***, short** *n***);**

read reads up to *n* bytes of data from the file descriptor *fd* and writes them into *buffer*. The amount of data actually read may be less than that requested if **read** detects EOF. The data are read beginning at the current seek position in the file, which was set by the most recently executed **read** or **lseek** routine. **read** advances the seek pointer by the number of characters read.

With a successful call, **read** returns the number of bytes read. Thus, zero bytes signals the end of the file. It returns -1 if an error occurs, such as bad file descriptor, bad *buffer* address, or physical read error.

Example

For an example of how to use this function, see the entry for **open**.

See Also

extended miscellaneous, fread

Notes

read is a low-level call that passes data directly to MS-DOS. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

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

read-only memory — Definition

As its name suggests, **read-only memory,** or ROM, is memory that can be read but not overwritten. It most often is used to store material that is used frequently or in key situations, such as a language interpreter or a boot routine.

See Also

Definitions, random access

realloc() — General utility (libc)

Reallocate dynamic memory
#include <stdlib.h>
void *realloc(void ***_ptr_, size_t** _size_**);**

realloc reallocates a block of memory that had been allocated with the functions **calloc** or **malloc**. This function is often used to change the size of a block of allocated memory.

ptr points to the block of memory to reallocate. If *ptr* is set to NULL, then **realloc** behaves exactly the same as **malloc**: it allocates the requested amount of memory and returns a pointer to it. *size* is the new size of the block. If *size* is zero and *ptr* is not NULL, then the memory pointed to is freed.

realloc returns a pointer to the block of *size* bytes that it has reallocated. The pointer it returns is aligned for any type of object. If it cannot reallocate the memory, it returns NULL. It calls **abort** if it discoveres that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block.

Example

This example concatenates two strings that had been created with **malloc**.

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

char *
combine(char **a, char **b)
{
      if(NULL == *a) {
            *a = *b;
            *b = NULL;
            return(*a);
      }
      else if(NULL == *b)
            return(*a);

      if((*a = realloc(*a, strlen(*a) + strlen(*b))) == NULL)
            return(NULL);
      return(strcat(*a, *b));
}

/* Copy a string into a malloc'ed hole. */
char *
copy(char *s)
{
      size_t len;
      char *ret;
```

```
      if(!(len = strlen(s)))
            return(NULL);
      if((ret = malloc(len)) == NULL)
            return(NULL);
      return(strcpy(ret, s));
}

main(void)
{
      char *a, *b;

      a = copy("A fine string. ");
      b = copy("Another fine string. ");

      puts(combine(&a, &b));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.3.4
The C Programming Language, ed. 2, p. 252

See Also

alignment, arena, calloc, free, general utility, lrealloc, malloc

Notes

If *size* is larger than the size of the block of memory that is currently allocated, the value of the pointer that **realloc** returns is indeterminate — it may point to the old block of memory, or it may not. If it is not, the contents of the old block of memory is copied to the new block.

record — Definition

A **record** is a set of data of a fixed length that has been given a unique identifier, and whose structure conforms to an exact description. An example of a record is an entry in a file of names and addresses: each entry has a fixed length, is marked by a unique identifier, and has a fixed number of bytes set aside in fixed order to record name, address, city, state, and ZIP code.

What is called a "record" in Pascal is called a "structure" in C.

See Also

Definitions, field, struct

register — C keyword

Quick access required
register *type identifier*

The storage-class specifier **register** declares that *identifier* is to be accessed as quickly as possible. **Let's C** will keep it in a machine register, if one is available.

It is not permissible to take the address of an object declared with the **register** designator, regardless of whether the implementation stores such an object in a machine register or not.

Example

For an example of using this specifier in a program, see **srand**.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 83

See Also

storage-class identifiers

Notes

An implementation must document how it handles variables declared to be **register**. Practice currently ranges from ignoring register declarations completely, to allowing a few register declarations for objects of an appropriate type (typically integer or pointer), to ignoring the designator and implementing a full global register allocation scheme.

register — Definition

A **register** is special high-speed memory within a microprocessor that can be addressed concisely and within which data can be stored and modified. The size and the configuration of a microprocessor's registers affect its computing potential. Registers can be manipulated much faster than RAM.

The routines in the **Let's C** libraries generally assume that they have been called from C programs. Thus, they may freely overwrite any registers that the compiler overwrites in its generated code. Thus, for the i8086, a library routine that returns **int** returns its value in AX, and preserves SI, DI, BP; in SMALL model, it will also preserve DS and ES. It can freely overwrite BX, CX, DX; in LARGE model it will also overwrite DS and ES.

See Also

Definitions

remove() — STDIO (libc)

Remove a file
#include <stdio.h>
int remove(const char ***filename***);**

remove breaks the link between between *filename* and the actual file that it represents. In effect, it removes a file. Thereafter, any attempt to use *filename* to open that file will fail. It is equivalent to the function **unlink**.

If you attempt to remove a file that is currently open, **remove** will fail. **remove** returns zero if it could remove *filename*, and nonzero if it could not.

Example

This example removes the file named on the command line.

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
      if(argc != 1) {
            fprintf(stderr, "usage: remove filename\n");
            exit(EXIT_FAILURE);
      }

      if(remove(argv[1])) {
            perror("remove failed");
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

LEXICON

Cross-references

Standard, §4.9.4.1
The C Programming Language, ed. 2, p. 242

See Also

file operations, rename, tmpfile, tmpnam

rename() — STDIO (libc)

Rename a file
#include <stdio.h>
rename(const char ******old***; const char** ******new***);**

rename changes the name of a file, from the string pointed to by *old* to the string pointed to by *new*. Both *old* and *new* must point to a valid file name. If *new* points to the name of a file that already exists, the old file is replaced by the file being renamed.

rename returns zero if it could rename the file, and nonzero if it could not. If **rename** could not rename the file, its name remains unchanged.

Example

This example renames the file named in the first command-line argument to the name given in the second argument.

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

main(int argc, char *argv[])
{
      if(argc != 3) {
            fprintf(stderr, "usage: rename from to\n");
            exit(EXIT_FAILURE);
      }

      if(rename(argv[1], argv[2])) {
            perror("rename failed");
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.4.2
The C Programming Language, ed. 2, p. 242

See Also

remove, STDIO, tmpfile, tmpnam

Notes

rename will fail if the file it is asked to rename is open, or if its contents must be copied in order to rename it.

return — C keyword

Return to calling function
return;
return *expression***;**

return is a statement that forces a function to return immediately to the function that called it.

return may also evaluate *expression* and pass its value to the calling function; the calling function regards this value as the value of the called function.

return can return a value to the calling function only if the called function was *not* declared to have a return type of **void**. The calling function is, of course, free to ignore the value **return** hands it.

If the called function is declared to return a type other than what **return** is actually returning, the value passed by **return** will be altered to conform to what the function was declared to return. For example,

```
main(void)
{
      printf("%s\n", example());
}

char *example(void)
{
      return "This is a string";
}
```

the pointer returned by **example** will be changed to an **int** before being returned to **main**. This is because **example** is declared implicitly within **main**, and a function that is declared implicitly is assumed to return an **int**. In environments where an **int** and a pointer are the same length, this code will work correctly. However, it will fail in environments where an **int** and a pointer have different lengths.

A function may have any number of **return** statements within it; however, a function can **return** only one value to the function that called it.

Reaching the last '}' in a function is equivalent to calling **return** without an expression.

Cross-references

Standard, §3.6.6.4
The C Programming Language, ed. 2, p. 70

See Also

break, C keywords, continue, goto, statements

Notes

If a program uses what is returned by a function as a value, and that function uses **return** without an expression, the behavior of the program is undefined.

rewind() — STDIO (libc)

Reset file-position indicator
#include <stdio.h>
void rewind(FILE ******fp***);**

rewind resets the file-position indicator to the beginning of the file associated with stream *fp*. It is equivalent to:

```
(void)fseek(fp, 0L, SEEK_SET);
```

rewind, unlike **fseek**, clears the error indicator for *fp*.

In previous releases of **Let's C**, **rewind** returned an **int**. It now returns nothing. This change was made to conform with the ANSI Standard, and may force some code to be rewritten.

LEXICON

Cross-references

Standard, §4.9.9.5
The C Programming Language, ed. 2, p. 248

See Also

fgetpos, file positioning, fseek, fsetpos, ftell

rindex() — Extended function (libc)

Find a character in a string
char ***rindex(char** **string*, **char** *character*);

rindex scans *string* for the last occurrence of *character*. If it finds *character*, it returns a pointer to it. If **rindex** does not find *character*, it returns NULL.

rindex is equivalent to the ANSI function **strchr**.

Example

This example uses **rindex** to help strip a sample file name of the path information. The path-name separator is '\'. The separator must be doubled so that **Let's C** will not interpret it as introducing an escape character.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PATHSEP '\\'          /* path name separator */

char *
basename(char *path)
{
      char *cp;
      return (((cp = rindex(path, PATHSEP)) == NULL)
            ? path : ++cp);
}

main(void)
{
      char *testpath = "A:\\foo\\bar\\baz";

      printf("Before massaging: %s\n", testpath);
      printf("After massaging: %s\n", basename(testpath));
      return(EXIT_SUCCESS);
}
```

See Also

extended miscellaneous, index, memchr, strrchr

Notes

This function is identical to the ANSI function **strrchr**. It is recommended that you use **strrchr** instead of **rindex** so that your programs will more closely approach strict conformity with the Standard.

runtime startup — Overview

The C runtime startup is a routine that is linked with a C program as the first part of an executable program. It performs the tasks needed to start and terminate the C environment. To begin the program, it initializes the stack and calls **main**; to conclude the program, it calls **exit** with the return value from **main**.

Let's C includes the following runtime startup routines:

crts0xs.obj	SMALL model
crts0xl.obj	LARGE model

The runtime startups used with the option **-VCSD** generate executable files that can be debugged with the Mark Williams C source debugger **csd**.

All of the above routines call **_main**. Depending upon which options you use to the **cc** command, these routines in turn can call one or more of the following object modules:

csdxl.obj	LARGE model, debug information for **csd**
csdxs.obj	SMALL model, debug information for **csd**
fxl.obj	LARGE model, floating point/8087 sensing
fxs.obj	SMALL model, floating point/8087 sensing
fxl87.obj	LARGE model, floating point/8087 only
fxs87.obj	SMALL model, floating point/8087 only
naxl.obj	LARGE model, no arguments to command line
naxs.obj	SMALL model, no arguments to command line
nsxl.obj	LARGE model, no STDIO in executable
nsxs.obj	SMALL model, no STDIO in executable
wxl.obj	LARGE model, wildcards on command line
wxs.obj	SMALL model, wildcards on command line

See the Lexicon entry for **cc** for more information on the no STDIO (**-ns**) and wildcards (**-w**) options.

See Also

Environment, exargs, execall, function call, _main

Notes

Source code is included for some of the runtime startup routines. Note, however, that this code should be edited only by experienced systems programmers.

rvalue — Definition

An *rvalue* is the value of an expression. The name comes from the assignment expression **E1=E2;** in which the right operand is an rvalue.

Unlike an lvalue, an rvalue can be either a variable or a constant.

Although the term "rvalue" is commonly used among programmers, the Standard prefers the term "value of an expression".

Cross-references

The C Programming Language, ed. 2, pp

See Also

Definitions, lvalue

Notes

All non-void expressions have an rvalue.

LEXICON