Subsections

Integer Functions, Random Number, String Conversion, Searching and Sorting: `<stdlib.h>`

To use all functions in this library you must:

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

There are three basic categories of functions:

- Arithmetic
- Random Numbers
- String Conversion

The use of all the functions is relatively straightforward. We only consider them briefly in turn in this Chapter.

Arithmetic Functions

There are 4 basic integer functions:

```
int abs(int number);
long int labs(long int number);

div_t div(int numerator,int denominator);
ldiv_t ldiv(long int numerator, long int denominator);
```

Essentially there are two functions with integer and long integer compatibility.

abs
functions return the absolute value of its `number` arguments. For example, abs(2) returns 2 as does abs(-2).

div
takes two arguments, `numerator` and `denominator` and produces a quotient and a remainder of the integer division. The `div_t` structure is defined (in `stdlib.h`) as follows:

```
typedef struct {
        int  quot; /* quotient */
        int  rem;  /* remainder */
} div_t;
```

(`ldiv_t` is similarly defined).

Thus:

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

int num = 8, den = 3;
div_t ans;


ans = div(num,den);
```

```
printf("Answer:\n\t Quotient = %d\n\t Remainder = %d\n", \
ans.quot,ans.rem);
```

Produces the following output:

```
Answer:
                Quotient = 2
  Remainder = 2
```

Random Numbers

Random numbers are useful in programs that need to simulate random events, such as games, simulations and experimentations. In practice no functions produce truly random data -- they produce ***pseudo-random*** numbers. These are computed form a given formula (different generators use different formulae) and the number sequences they produce are repeatable. A ***seed*** is usually set from which the sequence is generated. Therefore is you set the same seed all the time the same set will be be computed.

One common technique to introduce further randomness into a random number generator is to use the time of the day to set the seed, as this will always be changing. (We will study the standard library time functions later in Chapter 20).

There are many (pseudo) random number functions in the standard library. They all operate on the same basic idea but generate different number sequences (based on different generator functions) over different number ranges.

The simplest set of functions is:

```
int rand(void);
void srand(unsigned int seed);
```

`rand()` returns successive pseudo-random numbers in the range from 0 to (2^15)-1.
`srand()` is used to set the seed. A simple example of using the time of the day to initiate a seed is via the call:

```
srand( (unsigned int) time( NULL ));
```

The following program `card.c` illustrates the use of these functions to simulate a pack of cards being shuffled:

```
/*
** Use random numbers to shuffle the "cards" in the deck.  The second
** argument indicates the number of cards.  The first time this
** function is called, srand is called to initialize the random
** number generator.
*/
#include <stdlib.h>
#include <time.h>
#define TRUE    1
#define FALSE   0

void shuffle( int *deck, int n_cards )
{
        int     i;
        static  int     first_time = TRUE;

        /*
        ** Seed the random number generator with the current time
        ** of day if we haven't done so yet.
        */
        if( first_time ){
                first_time = FALSE;
                srand( (unsigned int)time( NULL ) );
        }

        /*
        ** "Shuffle" by interchanging random pairs of cards.
        */
```

```
        for( i = n_cards - 1; i > 0; i -= 1 ){
                int     where;
                int     temp;

                where = rand() % i;
                temp = deck[ where ];
                deck[ where ] = deck[ i ];
                deck[ i ] = temp;
        }
}
```

There are several other random number generators available in the standard library:

```
double drand48(void);
double erand48(unsigned short xsubi[3]);
long lrand48(void);
long nrand48(unsigned short xsubi[3]);
long mrand48(void);
long jrand48(unsigned short xsubi[3]);
void srand48(long seed);
unsigned short *seed48(unsigned short seed[3]);
void lcong48(unsigned short param[7]);
```

This family of functions generates uniformly distributed pseudo-random numbers.

Functions drand48() and erand48() return non-negative double-precision floating-point values uniformly distributed over the interval `[0.0, 1.0)`.

Functions lrand48() and nrand48() return non-negative long integers uniformly distributed over the interval `[0, 2**31)`.

Functions mrand48() and jrand48() return signed long integers uniformly distributed over the interval `[-2**31, 2**31)`.

Functions srand48(), seed48(), and lcong48() set the seeds for drand48(), lrand48(), or mrand48() and one of these should be called first.

Further examples of using these functions is given is Chapter 20.

String Conversion

There are a few functions that exist to convert strings to integer, long integer and float values. They are:

`double atof(char *string)` -- Convert string to floating point value.
`int atoi(char *string)` -- Convert string to an integer value
`int atol(char *string)` -- Convert string to a long integer value.
`double strtod(char *string, char *endptr)` -- Convert string to a floating point value.
`long strtol(char *string, char *endptr, int radix)` -- Convert string to a long integer using a given radix.
`unsigned long strtoul(char *string, char *endptr, int radix)` -- Convert string to unsigned long.

Most of these are fairly straightforward to use. For example:

```
char *str1 = "100";
char *str2 = "55.444";
char *str3 = "      1234";
char *str4 = "123four";
char *str5 = "invalid123";

int i;
float f;

i = atoi(str1);  /* i = 100 */
f = atof(str2);  /* f = 55.44 */
i = atoi(str3);  /* i = 1234 */
i = atoi(str4);  /* i = 123 */
```

```
i = atoi(str5);  /* i = 0 */
```

Note:

- Leading blank characters are skipped.
- Trailing illegal characters are ignored.
- If conversion cannot be made zero is returned and `errno` (See Chapter 17) is set with the value `ERANGE`.

Searching and Sorting

The `stdlib.h` provides 2 useful functions to perform general searching and sorting of data on any type. In fact we have already introduced the `qsort()` function in Chapter 11.3. For completeness we list the prototype again here but refer the reader to the previous Chapter for an example.

The `qsort` standard library function is very useful function that is designed to sort an array by a ***key*** value of ***any type*** into ascending order, as long as the elements of the array are of fixed type.

qsort is prototyped (in `stdlib.h`):

```
void qsort(void *base, size_t num_elements, size_t element_size,
   int (*compare)(void const *, void  const *));
```

Similarly, there is a binary search function, `bsearch()` which is prototyped (in `stdlib.h`) as:

```
void *bsearch(const void *key, const void *base, size_t nel,
     size_t size, int (*compare)(const  void  *,  const  void *));
```

Using the same `Record` structure and `record_compare` function as the `qsort()` example (in Chapter 11.3):

```
typedef struct {
        int   key;
                                                        struct other_data;
} Record;

int record\_compare(void const *a, void  const *a)
  {  return ( ((Record *)a)->key - ((Record *)b)->key );
  }
```

Also, Assuming that we have an `array` of `array_length Record`s suitably filled with date we can call `bsearch()` like this:

```
Record key;
Record *ans;

key.key =  3; /* index value to be searched for */
ans = bsearch(&key, array, arraylength, sizeof(Record), record_compare);
```

The function `bsearch()` return a pointer to the field whose key filed is filled with the matched value of `NULL` if no match found.

Note that the type of the `key` argument **must** be the same as the array elements (`Record` above), even though only the `key.key` element is required to be set.

Exercises

Exercise 12534

Write a program that simulates throwing a six sided die

Exercise 12535

Write a program that simulates the UK National lottery by selecting six different whole numbers in

the range 1 - 49.

Exercise 12536

Write a program that read a number from command line input and generates a random floating point number in the range 0 - the input number.

Dave Marshall
1/5/1999