

This document details the most commonly used standard library functions in some of the most commonly included header files.

## **Character Handling Functions**

The character handling built-in functions are all contained in the header file:

```
cctype
```

We include this by using the include directive as indicated below:

```
#include <cctype>
```

These functions return non-zero values (i.e. true) if and only if the value of the argument conforms to that required in the parameters of the function.

Some of the most commonly used character functions include:

<b>Function</b>	<b>Return Value</b>
int islower (int c)	Returns a non-zero value if a given character is a lowercase letter. i.e. a-z.
int isupper (int c)	Returns a non-zero value if a given character is an uppercase letter. i.e. A-Z.
int isalpha (int c)	Returns a non-zero value if a given character is a letter. i.e. A-Z, a-z.
int isdigit (int c)	Returns a non-zero value if a given character is a digit. i.e. 0-9.
int isxdigit (int c)	Returns a non-zero value if a given character is a hexadecimal digit. i.e. 0-9, A-F.
int isalnum (int c)	Returns a non-zero value if a given character is an alphanumeric character.
int isprint (int c)	Returns a non-zero value if a given character is a printable character. i.e. A-Z, a-z, 0-9, !, @, #, \$, %, etc.
int tolower (int c)	Converts a given character to lower case.
int toupper (int c)	Converts a given character to upper case.

## **Functions to Format Output**

C++ includes some very useful functions to help with the formatting of program output. Many of these functions are contained in the header file :

```
iomanip
```

We include this by using the include directive as indicated below:

```
#include <iomanip>
```

Some of the functions in this library include:

Function	Description
setw	Sets the width of the next field of output. If the size passed to the setw function is less than the size of the field, then the setw is ignored.
setprecision	Tells how many decimal places to display when outputting floating point numbers.
setiosflags ( <i>flags</i> )	A function that can be used to change the justification of data, the format of floating point data, and a whole range of other settings. . (See <i>flags</i> below).
setf ( <i>flags</i> )	A stream function which is equivalent to <i>setiosflags</i> , but is invoked differently.
precision	A stream function which is equivalent to <i>setprecision</i> , but is invoked differently.

Some of the flags available for use with *setf* and *setiosflags* include:

Flag	Description / Result
ios::showpoint	Always show a decimal point. e.g. 1.0 is displayed as 1.0 not 1.
ios::showpos	Display a leading + sign when the number is positive.
ios::fixed	Display floating point numbers in fixed format. e.g. 123.45
ios::scientific	Display floating point numbers in exponential format. e.g. 1.2345e2
ios::left	Left-Justify output
ios::right	Right-Justify output
ios::dec	Display numeric output in decimal (base 10) format.
ios::oct	Display numeric output in octal (base 8) format.
ios::hex	Display numeric output in hexadecimal (base 16) format.

By default in C++ displays numbers right-justified and strings are displayed left-justified. However, by using the above functions and flags we can change this.

As an example of using these functions and flags, consider this example:

```
cout << 20 << "Hi there!" << 123.4567 << endl;
```

This would display the following to the screen:

```
20Hi there!123.4567
```

We can use *setw* to space the output data into neat columns, as follows:

```
cout << setw(5) << 20 << setw(15) << "Hi there !" << setw(15)
<< 123.4567 << endl;
```

This would display the following to the screen:

```
20 Hi there! 123.4567
```

We can change the number of decimal places being displayed to two (or whatever we like) by using *setprecision*, override the default justifications of strings and numbers to make them all left justified, ensure that we display a decimal point for floating point numbers, and ensure that we output floating point numbers in fixed format, as follows :

```
cout << setprecision (2);
cout << setiosflags (ios showpoint);
cout << setiosflags (ios left);
cout << setiosflags (ios fixed);
cout << setw(5) << 20 << setw(15) << "Hi there !" << setw(15)
    << 123.4567 << endl;
```

Or, we could do the same thing using less *cout* statements, as follows :

```
cout << setprecision (2) << setiosflags (ios::showpoint)
    << setiosflags (ios::left) << setiosflags (ios::fixed);

cout << setw(5) << 20 << setw(15) << "Hi there !" << setw(15)
    << 123.4567 << endl;
```

And, in fact, we could do the same thing using less *setiosflags* function calls by combining the parameters using a Bitwise OR (*|*) operator :

```
cout << setprecision (2) << setiosflags (ios::showpoint | ios::left |
ios::fixed);

cout << setw(5) << 20 << setw(15) << "Hi there !" << setw(15)
    << 123.4567 << endl;
```

We can also achieve the same things using the stream versions of the functions. That is, *precision* instead of *setprecision* and *setf* instead of *setiosflags*. To call these stream functions, we use a dot notation with *cout* as follows :

```
cout.precision (2);
cout.setf (ios::showpoint | ios::left | ios::fixed);
cout << setw(5) << 20 << setw(15) << "Hi there !" << setw(15)
    << 123.4567 << endl;
```

## **Time and Date Functions**

Time and date built-in functions are contained in the header file:

```
ctime
```

We include this by using the include directive as indicated below:

```
#include <ctime>
```

Built into this library, are several special data types and structures, which are capable of representing time and date information.

Function	Description
time_t time (time_t *timer)	Returns the current date/time, in elapsed seconds since 00:00:00 GMT, January 1, 1970.
tm* localtime (time_t* timer);	Returns the current date/time as a date/time structure.
char* ctime (const time_t *timer)	Converts the time to a string containing the actual date and time.
char* asctime (const struct tm *timeptr)	Converts the time to a string containing the actual date and time.

### Random Number Generation

The mathematical built-in functions relating to random number generation are all contained in the header file:

`cstdlib`

We include this by using the include directive as indicated below:

```
#include <cstdlib>
```

Random numbers are very useful for a wide range of areas, including graphics, scientific, and games programming. Random numbers are useful because they can allow your program to act differently or produce different results based on the value of a random number.

Some of the most commonly used random number functions include:

Function	Description / Return Value
void srand (unsigned x)	Prepares (seeds) the random number generator for use. This function should be called first, before any other random number functions.
int rand (void)	Generates a random number between 0 and RAND_MAX
int random (int num)	Generates a random number between 0 and num-1

### **rand()**

The C++ standard library includes a pseudo random number generator for generating random numbers.

To generate a random number we use the `rand()` function. This will produce a result in the range 0 to `RAND_MAX`, where `RAND_MAX` is a constant defined by the implementation.

The following code generates a number in the range 0 to `RAND_MAX`, displaying it to the screen.

```
#include <iostream> // For cin, cout.
#include <cstdlib>  // For rand, seed.

int main()
{
    cout << rand() << '\n';
    return 0;
}
```

### **RAND\_MAX**

The value of `RAND_MAX` varies between compilers and can be as low as 32767, which would give a range from 0 to 32767 for `rand()`. To find out the value of `RAND_MAX` for your compiler run the following small piece of code:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
    cout << "The value of RAND_MAX is " << RAND_MAX
        << endl;

    return 0;
}
```

`Rand()`, the pseudo random number generator produces a sequence of numbers that gives the appearance of being random, when in fact the sequence will eventually repeat and is predictable.

### **srand()**

We can seed the generator with the `srand()` function. This will start the generator from a point in the sequence that is dependent on the value we pass as an argument. If we seed the generator once with a variable value, for instance the system time, before our first call of `rand()` we can generate numbers that are random enough for simple use (though not for serious statistical purposes).

In our earlier example the program would have generated the same number each time we ran it because the generator would have been seeded with the

same default value each time. The following code will seed the generator with the system time then output a single random number, which should be different each time we run the program.

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int main()
{
    srand((unsigned)time(0));
    cout << rand() << endl;
}
```

Don't make the mistake of calling `srand()` every time you generate a random number; we only usually need to call `srand()` once, prior to the first call to `rand()`.

### Generating a number in a specific range

If we want to produce numbers in a specific range, rather than between 0 and `RAND_MAX`, we can use the modulo operator.

Statistically it is not the most accurate way to generate a range but it's the simplest and adequate for most general purposes.

If we use `rand()%n` we generate a number from 0 to `n-1`.

By adding an offset to the result we can produce a range that is not zero based: `1 + rand()%n`

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int main()
{
    srand((unsigned)time(0));
    // Generates random number between 1 and 26
    cout << 1 + rand()%26 << endl;
}
```

### Assigning `rand()` values to integers

The `rand()` function returns an integer, therefore we can easily assign its return value to an integer variable:

```
srand((unsigned)time(0));
int random_integer = rand();
```

## Increased Randomisation

For a better method you should prefer the following:

```
random_integer = 1 + int(26.0 * rand()/(RAND_MAX+1.0));
```

The first number (the one before int) specifies the lowest number to be generated and the next one (first number after the int), when added to the first number (one before the int) indicates the highest number to be generated.