

# Basics in Standard and File I/O

---

## TABLE OF CONTENT

Bare basic for reading standard I/O using C.....	2
Standard I/O (stdout) .....	2
File I/O .....	3
Simple Open – Read - Write .....	3
Redirect I/O .....	4
Bare basic in I/O using C++ .....	5
Standard I/O (std) .....	5
I/O manipulator: Field Width – setw(n) .....	6
File IO (C++ style).....	9
Simple Open – Read - Write .....	9
Important Error Checking.....	10
Know the error status BIT.....	11
Redirect I/O .....	12

## BARE BASIC FOR READING STANDARD I/O USING C

NOTE: The scope of the following simply shows very brief description for file I/O to allow you to start reading and writing to a text file, not meant to teach you how exactly the whole FILE- I/O mechanism works.

### STANDARD I/O (STDOUT)

Standard output in C standard. Need to include "stdio.h" which usually is included by default if you use an IDE.

- use *printf*
- from the example above :

```
printf ( "The area of this rectangle is : %d metres.\n", area ) ;
```

- Other display format strings

- %d** – print the value of a decimal (based 10) integer
- %5d** - print the decimal integer in five character positions, right justified.
- %-5d** - left justified.
- %c** - print a single character.
- %s** - print a sequence of character. (more on characters in later days)
- %f** - print the value of a signed, decimal, floating point number.
- %6.2f** - print the decimal floating point number in six character positions, right justified and to an accuracy of two decimal places

Samples:

```
#include <stdio.h>

void main()
{
    int abc = 97;
    printf("1(%5d)\n", abc);
    printf("2(%-5d)\n", abc);
    printf("3(%10c)\n", 'A');
    printf("4(%-10c)\n", 'A');
    printf("5(%20s)\n", "hello!");
    printf("6(%-20s)\n", "hello!");
    printf("7(%*d)\n", 10, abc);
    printf("8(%*c)\n", 10, abc);
    printf( "Hello World.\n" );
    printf ( "The area of this rectangle is : %d metres.\n",
    area ) ;

    return;
}
```

## FILE I/O

### *SIMPLE OPEN – READ - WRITE*

Basic steps:

1. Open the file.
2. Do all the reading or writing.
3. Close the file.

#### Open a file :

```
FILE filePointer = fopen (fname, openMode);
```

openMode: determines whether you are opening the file for:

- writing or reading or appending
- text or binary

Sample:

```
FILE infp;
infp = fopen("inputFilename.txt","r");
outfp = fopen("inputFilename.txt","w");

char data;

if ( (infp==NULL) {
    printf(stdout, "Can't open input file %s\n", "inputFilename.txt");
    exit(-1);
}
if ( (outfp==NULL) {
    printf(stdout, "Can't open output file %s\n", "outputFilename.txt");
    exit(-1);
}
}
```

#### Reading from a file:

e.g. Data in inputfilename.txt shows:

```
amy 100
joey 50
```

To get the data and store in memory:

```
char name[10]; int score; char data;

if ( (data = fgetc(infp)) != EOF) {
    printf("%c", data);
    // or
    // fprintf(stdout, "%c", data);
}
```

Writing into a file:

```
fprintf(outfp, "%c", data);
```

Naturally, you can write string, integer, etc. into a file. You can write binary too, but you must open the file as "wb", "write with binary" mode.

Close a file:

```
fclose(infp);
fclose(outfp);
```

## REDIRECT I/O

First, you should know available access mode:

In C: fopen (filename, mode);

mode	
"r"	read: Open file for input operations. The file must exist.
"w"	write: Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.
"a"	append: Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations ( <a href="#">fseek</a> , <a href="#">fsetpos</a> , <a href="#">rewind</a> ) are ignored. The file is created if it does not exist.
"r+"	read/update: Open a file for update (both for input and output). The file must exist.
"w+"	write/update: Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
"a+"	append/update: Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations ( <a href="#">fseek</a> , <a href="#">fsetpos</a> , <a href="#">rewind</a> ) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

To open a file as a *binary file*:

A "b" character has to be included in the *mode* string, e.g.: "rb", "wb", "ab", "r+b", "w+b", "a+b") or

To Redirect stdout to a file: e.g.

```
FILE *fp;
char name[50] = "c:\\temp\\T";
static char ct = 1;
sprintf_s(name, sizeof(name), "%s%d.txt", name, ct++);
freopen_s(&fp, name, "w", stdout);
if (fp == NULL)
    exit(-1);
```

To recover stdout:

```
freopen_s(&fp, "CONOUT$", "w+", stdout);
```

---

## BARE BASIC IN I/O USING C++

- Advantage; One big advantage is of this over C's-I/O is **"type-safe"**.

## STANDARD I/O (STD)

Don't forget to include the Header files

```
#include <iostream>    // std::cin, std::cout
#include <iomanip>      // for set width and special format
using namespace std;  // for standard input and output stream
```

<< : called *insertion* –

e.g. `cout << var` : insert information from variable name "var" to the output stream.

>> : called *extraction* -

e.g. `cin >> var` : read as extract information from input stream and put in variable name "var".

Sample 1:

```
int a,b;
string name;

cin >> name;
cin >> a;
cin >> b;
```

or ultimately, you can gather them on a same line, like:

```
cin >> a >> b;
```

Sample 2: with output

```
cout << "Hello, " << name << ", you have entered " << a << "and" << b;
```

### I/O MANIPULATOR: FIELD WIDTH – SETW(N)

Sample 1:

```
int width = 10;
float amount = 5.5;
cout << setw(width) << amount;
```

or

```
cout << setw(width) << amount << "\n";
```

or

```
cout << setw(width) << amount<< endl;
```

Note: if you do not include "using namespace std", you will need to reference with "std::", e.g.

```
std::cout << "Please, enter the amount: ";
```

Sample 2 - with loop:

```
const int max = 12;
const int width = 6;
for(int row = 1; row <= max; row++) {
    for(int col = 1; col <= max; col++) {
        cout << setw(width) << row * col;
    }
    cout << endl;
}
```

Notice how "setw(n)" controls the field width, so each number is printed inside a field that stays the same width regardless of the width of the number itself.

However, setw(n)" is volatile. "*setw(n)*" only works for a single subsequent variable. You must apply "setw(n)" for each variable.

### I/O manipulator: Justification in Field

Choices are left and right. Numbers are by default right-justified.

```
setiosflags(ios::left);
setiosflags(ios::right);
```

Sample :

```
cout << setiosflags(ios::left) << 5.5;
cout << resetiosflags(ios::right)<< 5.5;
```

// need to reset in order to return to default, i.e right-justified.

*I/O manipulator: Controlling Precision*

There are two IO stream base :

- Fixed : write floating-point values in fixed-point notation
- Scientific : write floating-point values in scientific notation.

Sample 1:

```
double x = 800000.0/81.0;
setprecision(5);
cout << setiosflags(ios::fixed) << x;
```

then change the format to scientific notation:

```
cout << setiosflags(ios::scientific) << x;
```

Sample 2:

```
double a = 3.1415926534;
double b = 1996.1;
double c = 5.2e-10;

cout.precision(5);

cout << "default:\n";
cout << a << '\n' << b << '\n' << c << '\n';

cout << '\n';

cout << "fixed:\n" << fixed;
cout << a << '\n' << b << '\n' << c << '\n';

cout << '\n';

cout << "scientific:\n" << scientific;
cout << a << '\n' << b << '\n' << c << '\n';
```

Again, compile and run. Pay attention to the display and how each is made differently. Then, experiment with variations.

*I/O manipulator: Choose a character to fill the leading spaces in a number*

Sample :

```
int m = 1, d = 2, y = 2015;
cout << setfill('0')
cout << setw(2) << m << '/' << d << '/' << y << endl;
```

Compile and run. Then, try the one below:

```
cout << setfill('0');
cout << setw(2) << m << '/'
<< setw(2) << d << '/'
<< setw(4) << y << endl;
```

## Number Bases other than 10

For base 8 and base 16:

```
unsigned long x = 64206;
cout << x
<< " in base 8 is \"" << oct << x << "\"
<< " and in base 16 is \"" << hex << x << "\" << endl;
```

This stream formatting for different bases also works for input:

```
int x;
cin >> hex >> x;
```

Skip the following example if you are a beginner:

```
string convBase(unsigned long v, long base)
{
    string digits = "0123456789abcdef";
    string result;
    if((base < 2) || (base > 16)) {
        result = "Error: base out of range.";
    }
    else {
        do {
            result = digits[v % base] + result;
            v /= base;
        }
        while(v);
    }
    return result;
}

int main()
{
    unsigned long x = 64206;
    cout << "Hex:      " << convBase(x,16) << endl;
    cout << "Decimal: " << convBase(x,10) << endl;
    cout << "Octal:   " << convBase(x,8) << endl;
    cout << "Binary:  " << convBase(x,2) << endl;
    return 0;
}
```

## IO Manipulators Summary <iomanip>

Header providing parametric manipulators: (<http://www.cplusplus.com/reference/iomanip/>)

setiosflags	setprecision	get_time
resetiosflags	setw	put_time
setbase	get_money	
setfill	put_money	



## FILE IO (C++ STYLE)

NOTE: The scope of the following simply shows very brief description for file I/O to allow you to start reading and writing to a text file, not meant to teach you how exactly the whole FILE- I/O mechanism works.

### *SIMPLE OPEN – READ - WRITE*

Basic steps:

1. Open the file.
2. Do all the reading or writing.
3. Close the file.

### Open a file :

```
#include <iostream>
```

```
#include <fstream>
#include <iostream>
using namespace std;

ifstream infp; // create input object
ofstream outfp; // create output object

infp.open(fname, ios::in); // open a file for input mode
outfp.open(fname, ios::out); // open a file for output mode
```

Sample:

```
ifstream infp;
infp.open("inputFilename.txt", ios::in);

if (!infp) {
    cerr << "Can't open input file " << "inputFilename.txt" << endl;
    exit(1);
}

outfp.open("outputFilename.tx", ios::out);

if (!outfp) {
    cerr << "Can't open output file " << "outputFilename.txt" << endl;
    exit(1);
}
```

Reading from and writing to a file:

e.g. Data in inputfilename.txt shows:

```
amy 100
joey 50
```

To get the data and store in memory:

```
char name[10]; int score;

while (!infp.eof()) {
    infp >> name >> score;
    outfp << name << " " << score << endl;
}
```

Or

```
while (infp >> username >> score) {
    ...
}
```

Close a file:

```
infp.close();
outfp.close();
```

Additional specifier in C++ ("*x*"), that can be appended to any "*w*" specifier (to form "*wx*", "*wbx*", "*w+*x**" or "*w+bx*"/"*wb+*x**"). This subspecifier forces the function to fail if the file exists, instead of overwriting it.

**IMPORTANT ERROR CHECKING**

Continue ONLY if is NOT ( end of file || any logical error || Read/writing error )

```
while ( !(infp.eof() || infp.fail() || infp.bad()) ) {
    infp >> name >> score;
    outfp << name << " " << score << endl;
}
```

### KNOW THE ERROR STATUS BIT

You can get the error status flag (if you know how bitwise operation works)

```
ifstream infp;
infp.open ("test.txt");

// check if the fail bit is set
if ( (infp.rdstate() & ifstream::failbit ) != 0 )
    cerr << "Error opening 'test.txt'\n";
```

From [www.cplusplus.com](http://www.cplusplus.com):

<u>iostate</u> value (member constants)	indicates	functions to check state flags				
		<u>good()</u>	<u>eof()</u>	<u>fail()</u>	<u>bad()</u>	<u>rdstate()</u>
<b>goodbit</b>	No errors (zero value <u>iostate</u> )	true	false	false	false	goodbit
<b>eofbit</b>	End-of-File reached on input operation	false	true	false	false	eofbit
<b>failbit</b>	Logical error on i/o operation	false	false	true	false	failbit
<b>badbit</b>	Read/writing error on i/o operation	false	false	true	true	badbit

## REDIRECT I/O

First, you should know available access mode:

```
open (filename, mode);
```

Where `filename` is a string representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

mode	
<code>ios::in</code>	Open for input operations.
<code>ios::out</code>	Open for output operations.
<code>ios::binary</code>	Open in binary mode.
<code>ios::ate</code>	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
<code>ios::app</code>	All output operations are performed at the end of the file, appending the content to the current content of the file.
<code>ios::trunc</code>	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

```
e.g.: ofstream myfile;
      myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

To read file data into a string: e.g.

```
string line;

ifstream infp;
infp.open(fname, "r");

while (infp >> line)
{
    ...
}
```

To redirect stdout to a file: e.g.

Note: method `rdbuf()` returns a pointer to the [stream buffer object](#) currently associated with the stream, such as `cout`, `cin`, etc.

```
streambuf *psbuf, *backup;
ofstream ofp;

ofp.open( "outfilename.txt");
backup = cout.rdbuf(); // backup cout's stream buffer;
psbuf = ofp.rdbuf(); // get file's internal stream buffer
cout.rdbuf(psbuf); // assign stream buffer to cout
cout << "this will be written to the file instead of stdout";
cout.rdbuf(backup); // restore- make sure you will recover the stdout
ofp.close();
```

another sample segment for writing multiple values:

```
for (i=0; i < 10; i++)  
    cout << i << ",\n" ;
```

To redirect variable values to a file: e.g.

```
int i;  
char s;  
ifstream infp;  
  
infp.open("outfilename.txt");  
  
while (infp >> i >> s )  
{  
    cout << i << endl;  
}
```

To redirect standard input/output to a string stream:

Std::stringstream : Stream class to operate on strings.

```
using namespace std  
  
stringstream buffer;  
stringstreambuf *old = cout.rdbuf(buffer.rdbuf());  
  
cout << "go to the string buffer instead" << endl;  
  
string text = buffer.str();  
    // text will now contain "go to the string buffer instead\n"
```