90. File I/O

Objectives

- Read a file
- Write to a file

As you know values in variables/objects are lost when you close a program. But there are times when you want to save their values. For instance if a player reaches a new high score, you might want to save his/her high score (and his/her name – or he/she would not be happy...) and then print that person's name and high score every time that game runs. In this set of notes, I'll show you how to save data into text files.

Writing to a File

Run this:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f("hw.txt", std::ios::out);

    f << "hello world" << std::endl;
    int x = 0;
    double y = 3.14;
    char z[] = "c++ rules";
    f << x << ' ' << y << std::endl;
    f << z << std::endl;

    f.close();

    return 0;
}
```

Now look for the file `"hw.txt"` (without the quotes of course!). If you're using MS VS .NET if should be in your project folder (in your solution folder). If you're using my linux virtual machine, it will be the folder/directory where you run your program. In general, it's where you run the executable.

The `f` in your code is actually a file object:

```
    std::ofstream f("hw.txt", std::ios::out);
```

The name of the file is of course `hw.txt`. The second argument `std::ios::out` tells C++ that you want to create a file object for writing (i.e., output). Of course the class of this object is `ofstream` and it's in the `std` namespace and the header file for this class is kept in `fstream.h` which is why we have to `#include <fstream>`.

Note that the `operator<<` is used to push contents to the file. So it's really easy: it's just like input and output since day one of programming. In fact the `std::cout` is also a file except that it's a file associated with the console window.

Exercise. First try to guess what the file `hw.txt` looks like when you run this program:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f("hw.txt", std::ios::out);
    f << "hello world ... again" << std::endl;
```

```
    int x = 0;
    double y = 3.14;
    f << 42 << ' ' << x << ' ' << x + 1 << std::endl;
    f << y << std::endl;
    f.close();

    return 0;
}
```

Next, check the file and verify your guess.

Note that in the exercise the previous `"hello world"` is wiped out. **Therefore opening a file for writing with `std::ios::out` will wipe out the previous content.**

Exercise. Run this program and verify that the previous `hw.txt` is wiped out:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f("hw.txt", std::ios::out);
    f << std::endl;
    f.close();

    return 0;
}
```

Reading From a File

First create a file `hw.txt` with the following data:

```
hello world
0 3.14
42
```

Now run this program:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream f("hw.txt", std::ios::in);

    int x;
    double y;
    char z[100];

    f >> z;
    std::cout << "z: " << z << std::endl;

    f >> z;
    std::cout << "z: " << z << std::endl;

    f >> x;
    std::cout << "x: " << x << std::endl;

    f >> y;
    std::cout << "y: " << y << std::endl;

    f >> y;
    std::cout << "y: " << y << std::endl;

    f.close();

    return 0;
}
```

Appending to a File

First create a file `hw.txt` with the following data:

```
hello world
0 3.14
42
```

Now run this:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f("hw.txt", std::ios::app);

    f << "hello world" << std::endl;
    int x = 0;
    double y = 3.14;
    char z[] = "c++ rules";
    f << x << ' ' << y << std::endl;
    f << z << std::endl;

    f.close();

    return 0;
}
```

Note that this is the same program as the first ... except for this:

```
...

    std::ofstream f("hw.txt", std::ios::app);

...
```

You are now opening the file in append mode.

If you run the program, you will see that contents is **appended** to the end of the original file.

Flushing the File and std::endl

The write operations you make to a file does not add contents to the file immediately. Your OS (operating system) actually provides the file operations and in many cases, the I/O (input/output) is buffered, meaning to say that the contents during I/O is temporarily kept in memory.

Why?

For performance reasons. Because the OS is in charge of deciding when's the best time to perform certain operations. In the case of a file, accessing a hard drive slows down the system. So the OS will keep the contents to be written in memory and wait till there's a certain amount to write to the hard drive before doing a real write to the hard drive. It's the same like you would keep a bunch of checks to cash in your wallet and visit your bank only once a week.

This means that if you write 42 to a file, the actual file on your harddrive might not have a 42 in it immediately.

You can however **force** the contents to be written **immediately** using the **flush** method in the file object like this:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f("hw.txt", std::ios::app);

    f << "hello world" << '\n';
    f.flush();

    int x = 0;
    double y = 3.14;
    char z[] = "c++ rules";

    f << x << ' ' << y << '\n';
    f.flush();

    f << z << '\n';
    f.flush();

    f.close();

    return 0;
}
```

This is helpful if you want to synchronize (a little better) the actions in the program with your file on the harddrive. For instance if you're writing a game, you might want to have lots of print statements for debugging. If the print statements are delayed and there's a program error, the program might crash before the output is sent, therefore making your

debugging difficult.

You can also flush your `std::cout`:

```
std::cout << "hello world";
std::cout.flush();
```

Note that you already know from day 1 of programming that another way to print a newline character to the console output is

```
std::cout << "hello world" << std::endl;
```

Of course you can also do that for a file.

```
f << "hello world" << std::endl; // f opened for
                                 // writing
```

Now it turns out that when you send `std::endl` to a file of `std::cout`, not only is a newline sent, in fact there's a flush. This means that

```
f << "hello world" << '\n';
f.flush()
```

Is really the same as

```
f << "hello world" << std::endl;
```

and

```
std::cout << "hello world" << '\n';
std::cout.flush();
```

is the same as

```
std::cout << "hello world" << std::endl;
```

The close operation also causes a flush to occur. Therefore you need not issue a flush when you're about to issue a close.

By the way, when you use `std::cout` you are using an output file too (in most cases the file is tied to your console window). This output file is usually called "stdout". When you do

```
std::cout << std::endl;
```

you are printing a newline **and then flushing your stdout**. This ensures that whatever you have just printed is printed immediately without delay. If you're debugging a program and there are memory leaks, frequently the you want to print print statements to help with your debugging. The problem is that the program can crash before what you are printing appears. So in such cases, you want to flush your stdout as frequently as possible. So if you're trying to verify if the program crashed in function f(), you should do this:

```
std::cout << "before calling f()" << std::endl;
f(); // might be buggy
std::cout << "after calling f()" << std::endl;
```

The File Cursor/Pointer

Recall that when you were reading the file

```
hello world
0 3.14
42
```

with this program

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f("hw.txt", std::ios::app);

    f << "hello world" << std::endl;
    int x = 0;
    double y = 3.14;
    char z[] = "c++ rules";
    f << x << ' ' << y << std::endl;
    f << z << std::endl;

    f.close();

    return 0;
}
```

The reading of data for each variable stops at a whitespace (space, tab, newline). The next read will continue where it previous stopped.

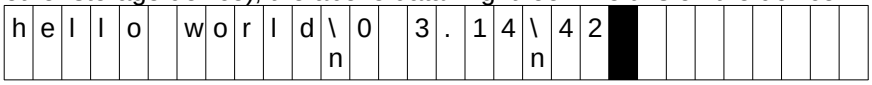
This tells you that each file remembers where the previous read stopped. This is called a file cursor.

When you open a file, the cursor points to the first character in the file. As you read, the cursor moves forward.

Let's look at the above file:

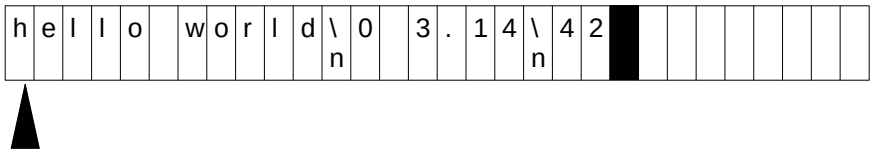
```
hello world
0 3.14
42
```

It looks like there are three "lines" in this file. Actually a file is made up of a bunch of bytes (or characters if you like). On the harddrive (or some other storage device), the above data might look like this on the device:



(It's actually a lot more complex than this but this is enough.)

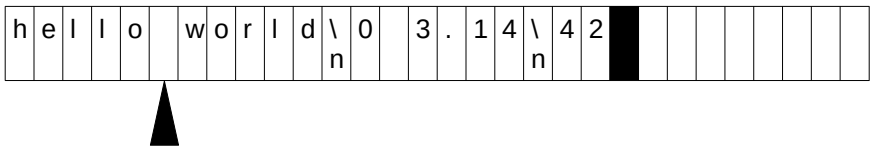
Each square in the above is a "slot" in your storage device to store a byte (or character) of data. You can think of your storage device as a pretty huge array if you like. The black square denotes the "end of the file" – EOF. Initially, when the file is opened, its file cursor points to the first byte, i.e.the character h:



If you execute this:

```
char x[100];
f >> x;
```

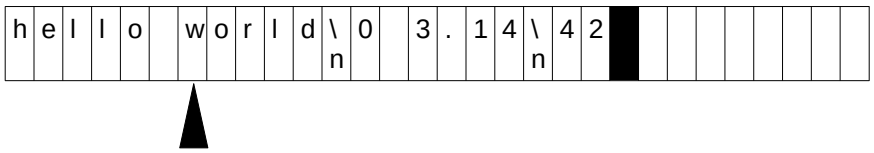
the characters starting from the character where the file cursor points to is read and put into array `x` until a whitespace is reached:



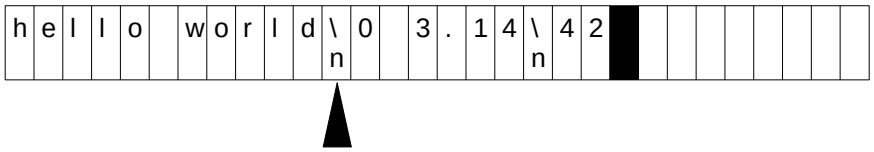
If you do this again:

```
f >> x;
```

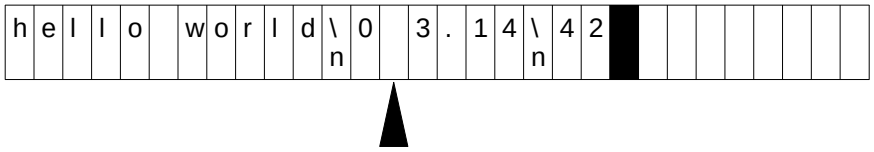
the file cursor will read and ignore whitespaces (spaces, tabs, newlines) until it sees a non-whitespace



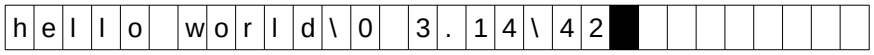
and then read the data, putting the characters into `x`, until it sees a whitespace:

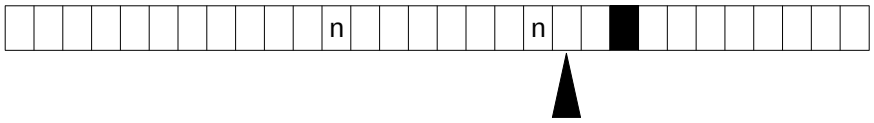


The next time you read the file and put the data into an integer variable, the integer 0 will be placed in the variable, and the pointer moves to this point:

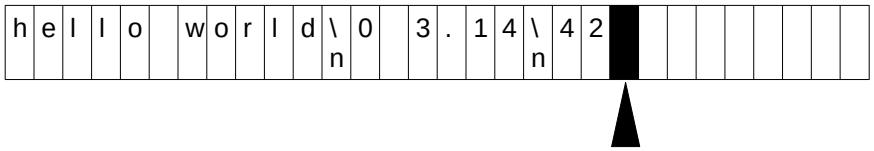


And after reading a `double` into a `double` variable it's here:





And finally after reading an `int` value it's here:



The file cursor has reached the end-of-file.

End-of-FIle

In all the previous examples up to this point, I know how much data there is to read. There are cases when you do not. For instance suppose you have a file of employee name and salary and there is no fixed number of employees. For instance on Monday it might be like this:

```
John 1234.56
Susan 1234.56
Tom 2345.67
Jane 2345.67
```

On Tuesday a new employee joins the company and it becomes this:

```
John 1234.56
Susan 1234.56
Tom 2345.67
Jane 2345.67
Harry 3456.78
```

In that case, you need to "continue to read till you reach the end-of-file". What happens is this ...

As you read the data from the file, the cursor moves. You can check if the cursor has moved just beyond the last readable data in the file. Let's try it.

First rewrite `hw.txt` so that it contains the following data:

```
2 3 5 7
11
13
17
19
23
29
```

Run this program:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream f("hw.txt", std::ios::in);

    while (!f.eof())
    {
        int x;
        f >> x;
        std::cout << "read ... x: " << x
                  << std::endl;
    }
    f.close();

    return 0;
}
```

`f.eof()` returns true if the file cursor has reached the **end-of-file**.

Oh no .... Errors!!!

In this section, I'll show you some of the ways to check for errors.

Many things can go wrong when it comes to file I/O (well ... many things can go wrong in general). For instance

- When you open the file ... it's not there!
- When you read/write to the file ... there's a hardware failure!
- When you read/write to the file ... the USB connection of the external HD was accidentally unplugged!
- When you write to the file ... the HD ran out of space!
- The file is corrupted and when you're reading input into an integer variable, the data contains 'z'!
- Etc!!!!!

Let's take it one at a time ...

Is the File Open?

The first thing to check is that when you open the file ... that it's actually opened! So first make sure that you don't have a file called xyz.txt, and next run this program:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream f("xyz.txt", std::ios::in);
    std::cout << f.is_open() << '\n';
    f.close();

    return 0;
}
```

First note that the program does not crash even though you attempted to open a non-existentent file. Closing a file that was not opened also did not generate a fatal program crash. We definitely did not see any unhandled exception thrown.

The method `is_open()` does return the `false` boolean value (that's why 0 was printed), telling you that the file cannot be opened.

Exercise. Now create the file `xyz.txt` and run the program again. Make sure 1 is printed.

So your program should look like the following (or something similar):

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream f("xyz.txt", std::ios::in);
    if (f.is_open())
    {
        std::cout << "error opening file xyz.txt"
                  << std::endl;
    }
    else
    {
        // do what you want to do ...
        f.close();
    }
    return 0;
}
```

Actually you can use the value of `f` to tell if the file is opened: if `f` cannot be opened, then the value of `f` is 0. Otherwise it will be a non-zero value. However using `is_open()` is a better idea. Anyway you can try this with

a non-existent `xyz.txt` and then where there's an `xyz.txt` file:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream f("xyz.txt", std::ios::in);
    if (!f)
    {
        std::cout << "error opening file xyz.txt" <<
std::endl;
    }
    else
    {
        // do what you want to do ...
        f.close();
    }
    return 0;
}
```

Exercise. Create an exception class for file opening error and rewrite the above code using the exception class.

Solution:

```
#include <iostream>
#include <fstream>

class FileOpenError
{};

int main()
{
    try
    {
        std::ifstream f("xyz.txt", std::ios::in);
        if (!f.is_open())
           throw FileOpenError();

        // do what you want to do ...
        f.close();
    }
    catch (FileOpenError & e)
    {
        std::cout << "error opening file!"
                  << std::endl;
    }

    return 0;
}
```

Or even this:

```
#include <iostream>
#include <fstream>


class FileOpenError
{};


class InputFile: public std::ifstream
{
public:
    InputFile(const char filename[])
      : std::ifstream(filename, std::ios::in)
    {
        if (!this->is_open())
           throw FileOpenError();
    }
};


int main()
{
    try
    {
        InputFile f("xyz.txt");

        // do what you want to do ...
        f.close();
    }
    catch (FileOpenError & e)
    {
        std::cout << "error opening file "
                  << std::endl;
    }

    return 0;
}
```

Read Error

Now suppose you've checked that the file is open for reading, what can happen?

You might be attempting reading an integer from file `f`:

```
int x;
f >> x;
```

when the contents from `f` to be processed is not an integer. For instance from the file pointer, the next sequence of data is `"hello"`. This is sometimes called a format error.

Such an error can be detected like this:

```
int x;
f >> x;
if (f.fail())
{
    std::cout << "x is garbage!!!\n";
    char y[100];
    f >> y;
    std::cout << "data read: " << y << '\n';
}
else
{
    // do something useful with x
}
```

In this case data is not lost. In other words, the file pointer does not move:

```
int x;
f >> x;
if (f.fail())
{
    std::cout << "x is garbage!!!\n";
    char y[100];
    f >> y;
    std::cout << "data read: " << y << '\n';
}
else
{
    // do something useful with x
}
```

There are other situations where error is more serious (example: hardware failure) where data can be lost. This is how you check for such errors:

```
int x;
f >> x;
if (f.bad())
{
    std::cout << "fatal error!!!\n";
}
```

```
else
{
    // do something useful with x
}
```

Such cases are really bad and recovery is difficult.

A detailed error handling code (assuming the file is already checked to be open) might look like this:

```
f >> x;
if (f.fail())
{
    // try to do some recovery, maybe read x in a
    // different way and proceed
}
else if (f.bad())
{
    // fatal error ... maybe print error message and
    // halt the program
}
else
{
    // everything is OK with reading x
}
```

If you don't care to handle `f.fail()` and `f.bad()` cases separately you can of course do this:

```
f >> x;
if (f.fail() || f.bad())
{
    // don't bother with recovery ... data was bad
    // anyway ... GIGO
    // just print an error message and halt the
    // program
}
else
{
    // everything is OK with reading x ... MOVE ON!!!
}
```

There's a shorthand for doing the above.
The opposite of `f.bad()` is `f.good()`. So you can do this if you like:

```
f >> x;
if (f.good())
{
    // everything is OK with reading x ... MOVE ON!!!
}
else
{
    // don't bother with recovery ... data was bad
    // anyway ... GIGO
    // just print an error message and halt the
    // program
```

```
}
```

So let's say you have a file containing first name, last name, monthly pay:

```
John Doe 1234.56
Jane Smith 2345.67
Harry Lee 3456.78
```

You can do this (assuming you have already checked that your file `f` is opened successfully):

```
double total = 0.0;
bool error = false;
while (!f.eof())
{
    char[100] fname;
    char[100] lname;
    double pay;
    f >> fname >> lname >> pay;
    if (f.good())
    {
        total += pay;
        std::cout << fname << ' ' << lname
                  << " is paid "
                  << pay << std::endl;
    }
    else
    {
        error = true;
        break;
    }
}
if (!error) // or use f.good()
{
    std::cout << "total: " << total << std::endl;
}
else
{
    std::cout << "DATA IN FILE IS CORRUPTED!!!"
              << std::endl;
}
```

Clearing the Error and Continuing

Once your file f hits an error, the error is stored in object f. If you do want to somehow continue with your processing of the file, you need to clear the error information stored in `f`. This is easy enough. You just do this:

```
f.clear();
```

Now why and when would you want to do this?

There are times when you cannot simply stop your program when you hit some corrupted data in your file. You have to attempt to guess how damaged is your file, and try to reach a point where you are passed the garbage so that your program can continue to do something useful.

For instance suppose a payroll file looks like this on Monday:

```
John Doe 1234.56
Jane Smith 2345.67
Harry Lee 3456.78
```

and on Tuesday someone from the HR department accidentally entered two new employees, forgetting that middle names should not be included in the file:

```
John Doe 1234.56
Jane Smith 2345.67
Harry Lee 3456.78
George E. Danger 4567.89
Susan Doyle 5678.90
```

The payroll program that prints out company checks might hit an error on line 4, printing out only checks for the first 3 employees. A smarter program will attempt to read past the corrupted line, and hope that the next line is correct. If so, this program will continue to print the check for Susan Doyle.

This is why we want to be able to clear the error information in the file object. We'll come back to this later when I've explained to you how to for instance read passed the current line of data.

C-style File I/O

So far I've shown you how to work with files using the C++ file streams. Of course C was the father of C++. How do C programmers work with file? There are actually two different file I/O libraries that C programmers (and also for C++ programmers since C++ programmers can also use C file I/O libraries too) can use: one comes from the Unix operating system and another from C language. Therefore there are actually three different ways to work with files:

- Unix file pointers
- C file streams
- C++ file streams (what I talked about in this set of notes)

Now why should you be interested in C-style file I/O? Because many low-level code is written in C. That includes Unix and various unix utilities. Low-level network programming (i.e. network socket programming) is also usually done in C too.

So you had better learn it!!! I will leave it to you read this up on your own. Make full use of the web and google (example google "c style file pointers").