**I/O Streams**

A **stream** is a flow of characters (or other kinds of data). If flow is in, input stream, if flow is out, output stream. I/O can be from keyboard, or from a file. e.g., *cin* and *cout* are streams.

We can define I/O streams, along with where the streams goes (come from). *cin* comes from keyboard, and *cout* goes to screen.

**6.1. Streams and Basic File I/O**

1. **File I/O**

In C++, a stream is an *object.* If you want to use a stream to get input from a file (or give output to a file), you must declare the stream and you must connect the stream to the file.

* The **type** for input-file and output-file stream variables are named ifstream and ofstream.
* These types are defined in the library #include <fstream>
* These types are also under namespace std.

**a) File Input**

|  |  |
| --- | --- |
| *ifstream* inFile ; or:  inFile.open ("file\_directory") ; | ifstream inFile ("file\_directory") |

Now, our program can take input from the file using the extraction operator >>. **Each time we call the operator, we move to the next data in the file.** **>> skips all blank spaces, and will stop after getting enough data for the datatype of the variable**

**This may lead to unexpected errors. For example:**

|  |  |  |  |
| --- | --- | --- | --- |
| inFile | main.cpp | int a | float b |
| 3.2  2.6 | …  inFile >> a;  inFile >> b; | 3 | **0.2** |

**b) File output**

|  |  |
| --- | --- |
| *ofstream* outFile ; or:  outFile.open ("file\_directory") ; | ofstream outFile ("file\_directory") |

* Open for writing (default)
  + If file already exist, it is first erased.
  + Otherwise it is newly created.
* Open for append
  + outFile.open (“data.txt”, ios::app) ;
  + Writing adds to the end of the file

Now, our program can send output to the file using the insertion operator <<. For example:

outstream << “oneNumber = “ << oneNumber << “anotherNumber = “ << anotherNumber ;

Notice that after the file has been opened for the first time, *the file is then referred to by naming the stream that is connected to the file in the program.*

+) Why do we need to open the file? What if we instead just do write("filename") or append("filename")

🡪 Performance reason. Everytime we read/write/append, etc. to a file, there is a **permission check** needed to occur. Therefore, instead of doing this permisison check everytime, we do it once while opening the file.

+) .open() returns file descripter (big data structure), containing info that we can operate on our file

🡪 Takes up a memory space.

🡪 Every file should be closed after working with, using the method .close()

inStream.close () ;

1. **Buffering**

Accessing Hard Drive/SSD is slower than accessing Memory (RAM), up to a couple thousand times. Therefore, if we do writing to files normally if we wait for the statement to actually hit the hard drive, and there will be a delay.

🡪 OS can do something smart to hide this slow latency from you. There is a space allocated in the memory by the OS, called a buffer, to act as an intermediate accessor. When we perform writing action, the statement will **immediately return**, before the actually data is written in the file.

The data on the buffer will be eventually written to the hard drive for safety during the operation. However, there are times where we might want the memory to be written directly to the hard drive (files), so we can use .flush() to force this action. **This will flush all data in the buffer of the specified outFile to the actual file.**

outFile.flush()

Note that even if after we call .flush(), this is only on the side of the OS, meaning it attempted to flush. There may be circumstances where the hard drive might not deliver. We have to accept that the writing and saving to file operation may sometimes fail.

**Therefore, what's the difference between \n and endl?**

**When using endl, we are telling the OS to flush (everything for that file) 🡪 Should use endl.**

**File Error Handling**

1. **File doesn't exist**

In case the file doesn't exist, or not opened correctly, .fail() returns true.

inStream.fail ()

As we add this statement to our program, we can **handle the error** detected:

* Return 1
* **Exit the program**, using the .exit(<Integer\_Value>) function (#include stdlib)

Any may be used, but by convention, 1 is used for a call to exit that is caused by an error, and 0 is used in other cases. (this will then be handled by the operating system)

1. **Cannot read (data mismatch)**

Say we define

int a

But the first character in the file is 'h'. This will also cause an error, and we can also catch will .fail()

**Summary of File I/O Statements**

Shape, arrow

Description automatically generated

using namespace std;

int main () {

ifstream inStream; ofstream outStream;

int a; float b;

inStream.open("infile.dat");

if (inStream.fail( )) { cout << "Input file opening failed." << endl; exit(1); }

outStream.open("outfile.dat");

if (outStream.fail( )) { cout << "Output file opening failed." << endl; exit(1); }

inStream >> a;

if (inStream.fail()) { cout << "Failed to read a" << endl; exit(1); }

inStream >> b;

if (inStream.fail()) { cout << "Failed to read b" << endl; exit(1); }

outStream << "a = " << a << " and b = " << b << endl;

inStream.close( );

outStream.close( );

return 0;

}

1. **Continue after error**

.ignore(int num\_of\_chars, char delimiter)

* Discard num\_of\_chars characters, up to first delimiter, **which ever comes first**.

.clear()

* **Clear the error state flag** (allows continuous reading of data)
* If we do not clear the error flag, the **inFile.fail() we always return true 🡪 stuck**

Example:

if (inFile.fail()) { return 1; }

while (1) {

inFile >> a;

if (inFile.fail()) {

cout << "failed.." << endl;

inFile.clear();

inFile.ignore(100, '\n');

continue;

}

cout << "a = " << a << endl;

break;

}

return 0;

1. **End of file** .eof()

We need to always add this flag to handle the error when we reaches the end of a file. Or else, we might be stuck at the end of the file.

🡪 We should code our while loop as:

while (!inFile.eof())

**6.2. Tools for Stream I/O**

1. **String Stream**

<< and >> operators

Based on the type of the variable, the program can "parse" the input to fit it.

🡪 stringstream allows you to do that!

Text, letter

Description automatically generated

stringstream acts like a buffer, which "stores" the string inputs from input streams and can parse the output to variables, either if we want to extract strings or integers 🡪 stringstream does the work for us, and make it easier to detect bugs.

For example, we can get input from the user, put into a stringstream called "ss", and then extract the information to saved variables.

string cin\_input;

getline (cin, cin\_input); // store user input into "cin\_input"

stringstream ss(cin\_input); // create a stringstream, adding cin\_input to it. We can also do

// ss << cin\_input;

string first, second;

int third;

ss >> first >> second >> third;

1. **Formatting Output with Flags**

The layout of a program’s output is called the **format** of the output. In C++ you can control the **format** with commands that determine such details as the number of spaces between items and the number of digits after the decimal point.

Recall the formula for outputting a significant number of decimal places 🡪 Output formatting function.

- First, you can use these formatting commands on any output stream 🡪 We can also apply the same commands to any defined output streams, to ensure the number of decimal points.

outStream.setf (ios::fixed);

outStream.setf (ios::showpoint);

outStream.precision(2);

- Every output stream has a member function named *precision*. When your program executes a call to *precision*, then from that point on in your program, any number with a decimal point that is output to **that stream (only)** will be written with a total of 2 digits after the decimal point.

- The first 2 calls are **formatting instructions**. The member function *setf* , stand for set **flags**. A **flag** is an instruction to do something in one of two possible ways. If a flag is given as an argument to setf, then the flag tells the computer to write output to that stream in some specific way.

+) The flag ios::fixed causes the stream to output numbers of type *double* in what is called the **fixed-point notation**. If the flag ios::fixed is set (by a call to setf), then all floating-point numbers (such as numbers of type double) that are output to that stream will be written in ordinary everyday notation, rather than e-notation.

+) The flag ios::showpoint tells the stream to always include a decimal point in floating-point numbers. Therefore, the output will include the decimal point even if all the digits after the decimal point are 0.

**Some common flags**

Text

Description automatically generatedText

Description automatically generated

To unset a flag, use unsetf outStream.unsetf (<flag>)

1. **Manipulator**

A manipulator is a function that calls a member function. Manipulators are placed after the insertion operator <<, as if it was an item to be output (recall << endl). They may or may not have arguments.

To use *setw* and *setprecision*, you must #include <iomanip>

*setw*

setw calls the member output formatting function .width(int). The width function tells the stream how many spaces to use when giving an item as output, and is **applied to only the next item that is output**.

*width* will create a number of blank spaces. The amount of blank spaces depends on the length of the next item. However, if length is more, no spaces will be added, but all output will be output.

Therefore, we can use the setw manipulator:

cout << "Start" << setw(4) << 10 << setw(4) << 20255 << setw(6) << 30;

🡪 Start\_\_1020255\_\_\_\_30

*setprecision*

This does exactly the same as member function .precision() (still needs ios::fixed and ios::showpoint)

cout.setf(ios::fixed);

cout.setf(ios::showpoint);

cout << "$" << setprecision(2) << 10.3 << endl << "$" << 20.5 << endl;

The settings stay until you reset to some other number by another call to either setprecision or precision

1. **Streams as Arguments to Functions**

Text, application

Description automatically generatedA stream can be an argument to a function. **The only restriction is that a stream parameter cannot be a call-by-value parameter.**

* 1. **Character I/O \*\*\***

All inputs and outputs are considered as characters (e.g., hardware is always reading ‘a’ and ‘b’, not ‘ab’)

🡪 C++ provides some low-level facilities for input and output of character data that includes no automatic conversions. You could then write IO functions that read and write numbers in Roman.

1. **Member Functions *get, put* and *putback***

- *get* allows program to read in **one character of input** and store in type *char*.

When we normally use >>, the skipping of blanks are done automatically. *get* is more manual.

The *get* takes one argument, which is a variable of type *char* that will receives the input.

char nextSymbol;

cin.get(nextSymbol);

Text

Description automatically generatedIt takes everything, including the ‘\n’. For example:

If we type in “ab<enter>”

c1 = ‘a’

c2 = ‘b’

c3 = ‘\n’

We can help avoid this by using a do-while loop:

char symbol;

do {

cin.get(symbol) ; 🡪 This will read everything, including blanks, but avoid <enter>

cout << symbol ; (using the >> operator will avoid this, as it ignores the blanks)

} while (symbol != '\n') ;

- *put* outputs 1 character to the screen. It takes 1 argument, which is a *char* to be outputted.

- When you want to avoid reading, let’s say a blank in your input stream, you have to read it in order to know what to avoid. However, after reading it, the character is now removed from the input stream

🡪 *putback* allows you to **read up to** a character, and then **putback this character to the input stream**.

We can putback whatever character we want into the input stream; however, the file **will NOT change.** Only our program will consider as if the input file has been changed.

Text

Description automatically generatede.g., Checking user input

Because ans is only a *char*, so if we don’t get excess character ‘o’ when user input ‘no’, the next number input will have the ‘o’

🡪 **Wrong**

1. **Making Stream Parameters Versatile**

If we want to include IO streams as parameters to functions, but wants to include **both IO streams from files and keyboards**, we define the function with argument of type *istream*/*ostream* (without the f)

🡪 We will then include the I/O stream in the function parameter when calling the function, which specifies where we get/give our input/output (keyboard/file and screen/file)

*Note:* Sometimes programmers would like to include a function that takes IO streams from keyboard only 🡪 They use **function overloading**, to include a function with no parameters that takes input from the keyboard.

Then, we can call these 2 functions and get the same results:

functionName (cin); functionName ();

**OR** use default argument as cin.

1. **Checking for the End of a File**

Method 1 while (inStream >> next) *Should be used with numeric input*

Say we write a program to take number inputs from a file, and return the average. The preferred way to test for the end of the file is

double next, sum = 0;

int count = 0;

while (inStream >> next){

sum = sum + next;

count++;

}

cout << sum / count;

Method 2 using the .eof () member function *Should be used with text input*

Every input-file stream has a member function called *eof* that can be used to determine when all the the file has been read and there is no more input left in the program (end-of-file).

The function takes no argument and returns a boolean value.

inStream.get(next);

while (! inStream.eof( )) {

cout << next; //or cout.put(next) ;

inStream.get (next) ;

}

*Note* that the eof() returns true **only when the program has attempted to read the next character.**

This end-of-file marker should not be tried to be written out. It is automatically placed by the system.

Text

Description automatically generated**Example**: Editing a Text file. The input file is “cad.dat” and output file is “cplusad.dat”. Assume that we want to copy input file to output, but replace “C” to become “C++”.

Note how we used .get() 2 times, because .eof() requires the file end mark to be already collected.

1. **Predefined Character Functions \*\*\*\***

Included in the library #include <cctype>

|  |  |  |  |
| --- | --- | --- | --- |
| toupper (char)  tolower (char) | isupper (char)  islower (char) | isalpha (char)  isdigit (char) | isspace (char)  isblank (char) |

*NOTE* that toupper() and tolower() actually returns ASCII value of character. Recall that *char* are stored in memory as these values. Therefore, cout << toupper (‘a’) will output 97

🡪 To ensure the computer is treating value returned as characters, we need to place value returned in variable of type *char*, **or** type case the returned value