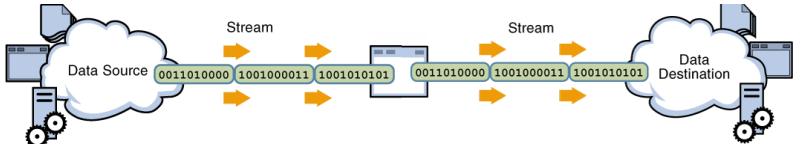
The Standard Streams

At the lowest level, all input and output is a stream of bytes flowing through your program. The bytes may come from a file, your keyboard or even from some remote computer on the network. Your program looks at the stream and changes it, consumes it, or sends it on to output.



Programs that process streams of characters are called **text** filters.

When you run a program, **where** do the input bytes **come from** and where do the **output bytes go to**? What is the data source (in the illustration above), and what is the data destination? Before your program starts, the **operating system** automatically opens three **standard** streams:

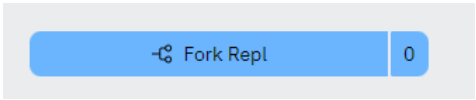
- `stdin` (standard input)
- `stdout` (standard output)
- `stderr` (standard error)

In C++, the built-in streams are used to initialize the `cin`, `cout` and `cerr` I/O objects. Java does the same thing, but uses them to initialize the `System.in`, `System.out`, and `System.err` objects. Python uses the streams directly.

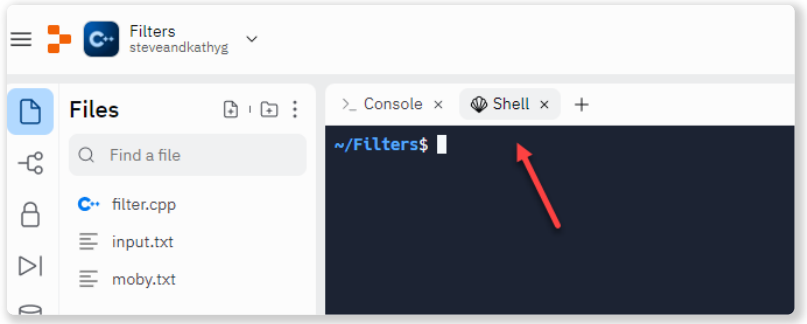
The operating system **connects these streams** to your console (screen and keyboard). But, **before** you **run your program**, you may ask the OS to connect each stream to a different endpoint. This is known as **redirection**.

The *cat* Filter

Click the "running man" to open a Replit project which already contains a few files. Click the **Fork Repl** button to get your own copy, and then let's look at a few built-in Unix filters.



Close the editor tab (which contains the **Makefile**) and then click the link for the Shell.



Type the following command in the shell (terminal), and press `ENTER`.

```
$ cat
```

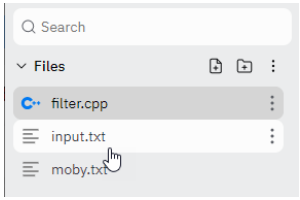
The cursor simply blinks; you **don't** get a new prompt. Go ahead and type a few lines of text, pressing `ENTER` at the end of each line. The **input you typed is echoed** on the next line. Press `CTRL+D` to return to the prompt.

- Filter programs **read from standard input** and **write to standard output**.
- The `cat` filter **concatenates** each input character to standard output. In Windows, the equivalent filter is named `type`.
- The filter **stops** reading when it reaches **end-of-file**. In Unix, you simulate that by typing `CTRL+D` from the terminal. In Windows, it is `CTRL+Z`.

A filter is **not meant to be run interactively**. Instead, it is meant **process a stream of data** that is supplied from a file, a network stream or some other source. The easiest way to supply such a stream is to use **input redirection**.

Input Redirection

Input redirection allows you to run a program, and have that program get its input from a file or device, instead of from the keyboard. Your program doesn't need to change at all; it still reads from `cin` as always.



To see how input redirection works, first, open the file named `input.txt` by clicking its tab, so you can see what it contains. Then, type this command in the shell:

```
$ cat < input.txt
```

The **input redirection symbol** `<` asks the operating system to first open `input.txt` and then to connect that to the **standard input** stream. Now, `cat` gets its input **from the file** instead of from the keyboard.

```
$ cat < input.txt
This is text stored in "input.txt".
A second line in input.txt
$
```

When all of the data has been processed, the prompt returns. (I've colored the output so that the input you type appears in teal, and the output from the command appears in blue.)

Output Redirection

All standard output streams are connected to the console; any output appears on your screen. You can **redirect standard output** by using the `>` symbol when you run:

```
$ cat < input.txt > output.txt
```

This time, no output will appear on your screen; instead, the file `output.txt` will be created and all the output, which would have been sent to the screen, will instead be written to the file. **This can be a little dangerous**, because if there is **already** an `output.txt`, it will be **overwritten** with the new data.

Ater you've typed the previous line, you can examine the new contents of `output.txt` by using `cat` again, like this:

```
$ cat < input.txt
This is text stored in "input.txt".
A second line in input.txt
$
```

Instead of erasing the existing data in the output file, you can **append** to it by using the `>>` symbols like this:

```
$ cat < input.txt >> output.txt
```

Try it and see what `output.txt` contains now.

Error Redirection

Type this command (exactly) in the shell, and press `ENTER`:

```
$ cat > output.txt < input.text
```

In this case, there **is no file** named `input.text`, so `output.txt` is **erased**. The `cat` filter prints an error message on the **standard error stream**, still connected to the screen. **Output redirection only redirects standard output, not standard error.**

You can the **redirect standard error** stream by using the symbol `2>` like this:

```
$ cat > output.txt 2> err.txt < input.text
$ cat < output.txt
$ cat < err.txt
bash: input.text: No such file or directory
$
```

Now `output.txt` is still empty, but `err.txt` contains the errors that originally appeared on the screen. **Combine both** into a single stream (which may be sent to a file) like this.

```
$ cat > combined.txt 2>&1 < input.text
```

Sometimes, you **don't want to see** either the error messages or any progress reports. For instance, if you try to remove a file which doesn't exist, the shell displays an error message like this:

```
$ rm filter.exe
rm: cannot remove 'filter.exe': No such file or directory
```

Instead of redirecting those messages to a file, you can send them to the **"bit bucket"** which has the name, (in Unix), `/dev/null`. (If you are using redirection on Windows, the name is `NUL:` with the trailing colon.) Anything redirected to `/dev/null` just disappears.

Pipes & Pipelines

Input redirection gets input from a file and output redirection sends its data to another file. Pipes, however, redirect the **output** of one program so that it acts as the input **of another program**. The pipe character is the **vertical bar**. Several pipe commands is called **a pipeline**.

The Unix ls command shows the files of the **current directory** on standard output.

```
$ ls
err.txt filter.cpp input.txt moby.txt output.txt
```

Of course you can save the directory listing to a file using output redirection:

```
$ ls > files.txt
```

However, instead of saving it, we can **pipe the output** to the `wc` (**word count**) filter, adding a command-line switch `-l`, to indicate that we only want to count the number of lines. Try it yourself and see what happens.

```
$ ls | wc -l
```

Here is another pipeline which lists the current directory, and then **sorts the output** in reverse order, sending that output to the screen.

```
$ ls | sort -r
```

One of the most useful Unix filters is `grep` (which stands for the mouthful "global regular expression parser"). While quite complicated, especially when used with **regular expressions**, it is easy to use for searching through text to find a particular word.

Let's find out, for instance, on which lines the name **Ishmael** is used in Moby Dick.

```
$ cat < moby.txt | grep "Ishmael" -n
```

And how many lines are there??

```
$ cat < moby.txt | grep "Ishmael" -n | wc -l
```

Built-in Filters

Here are ten of the most common (and useful) built-in Unix filter programs. To find out how to use the program, just type ***man command*** into Google search, replacing ***command*** with the name of the filter program.

1. `cat`: Displays the text of the file line by line.
2. `head`: Displays the first `n` lines of the specified text files. If the number of lines is not specified then by default prints first `10` lines.
3. `tail`: Works the same way as `head`, just in reverse order. The only difference in `tail` is, it returns the lines from bottom to up.
4. `sort`: Sorts the lines alphabetically by default but there are many options available to modify the sorting mechanism. Be sure to check out the *man* page to see everything it can do.
5. `uniq`: Removes duplicate lines. `uniq` only removes continuous duplicate lines. First use `sort` on your data before passing it to `uniq`.
6. `wc:` Prints the number of lines, words and characters in the data.
7. `grep:` Searches for particular information in a text file.
8. `tac`: The reverse of `cat`. Instead of printing from lines `1` through `n`, it prints lines `n` through `1`
9. `sed`: `sed` stands for **stream editor**. It allows you to apply search and replace operations on your data very effectively. `sed` is an advanced filter and all its options can be seen on its *man* page.
10. `nl`: `nl` is used to number the lines of your text data

Process Filters

Filters, you recall, are programs that read from standard input, and which write to standard output. Filters may change, use, or learn about the characters flowing through your program. Two kinds of filter programs are **process** filters and **state** filters.

- A process filter **does something to** the characters it encounters.
- A state filter **learns something about** the stream by examining characters.

Process filters **apply some basic rule**—the process—to the values in the stream. The simplest process filter is: ***read and echo*** (although I suppose that read and ignore would actually be simpler). That's what the `cat` filter does.

Process filters typically solve problems like this:

- Copy files or search for a particular value in a stream (`cp` and `grep`)
- Case modification or changing character order in a stream.
- Stream editing using a sequence of editing commands (`sed`)
- Translating data from one form to another (decimal to binary)

State Filters

State filters produce information by learning something about the data in a stream. State is shorthand for saying "what is the current status of this data". Characters, for instance, have values, but also belong to groups, like digit characters, alpha characters and so on.

State transitions are changes from one state to another. Most state filters work by finding the state transitions and then performing some action. Here are some uses:

- Counting the number of words in input (counting word transitions) (`wc`)
- Printing one sentence per line (looking for a period, question or exclamation mark)
- Compressing input (turn off echo when in blank-spaces state)

Often programs will contain both process-filter and state-filter portions. For one homework this week you'll write a state filter that removes comments in C++ source code, while in Lecture you'll write a process filter which encrypts and decrypts text.

Your First Filter

Now, let's look at writing our own filters. We'll continue working with the Replit project we used when learning about redirection. You can open it from your Repit account, or click the "running man", and re-fork it.

The program `filter.cpp` is the simplest possible version of the built-in filter `cat`. Remember that the `cat` command reads a character from **standard input** and sends it on to **standard output**, stopping only when there is no more input to be processed.

There are three ways to process input:

- Raw, or **unformatted input** (a byte or character at a time)
- **Line-oriented** input (one line at a time)
- **Formatted** or token-based input (a "word" at a time)

The program ***filter.cpp*** uses **raw, unformatted input**. Build and run the program in **Replit** like this. First, open the **Shell** tab and then type:

1. `make filter`. If you have only a single file, you can build it by giving make the name of the output file. The **make** program finds ***filter.cpp*** file and then compiles and links it.
2. `./filter < input.txt` to run the program. This should produce **exactly the same output** as using `cat`.

Note, in Unix, to run a program located in **the current working directory**, first type the directory `./` and then the name of the program. You can repeat any of the previous exercises replacing `cat` with `./filter`, and the results should be the same.

Data Loops

Let's look at how the *filter* program works. C++ input streams read a single character using the **member function** `cin.get()`. To **read successive characters**, until all of the data has been processed, use a **data** (or `eof-controlled`) loop. (`eof` is shorthand for **end-of-file)**.

```
while there is still data to process
  read a data item
  process the data item
```

Translate this into C++ by **using streams as conditions**, as shown here:

```
char ch;
while (cin.get(ch))
{
    cout.put(ch);   // print the output
}
```

The expression `cin.get(ch)` does two things.

1. **It reads** the next character from the stream into the `char` variable `ch` (which is passed to the function by reference). Whitespace **is not** skipped.
2. **It returns the input stream** (in this case `cin`) after reading the variable so you can determine whether the I/O operation succeeded.

The `cin` object has a member function, named `fail()`, which indicates whether the last operation succeeded. `fail()` **is implicitly called when a stream is used as a condition.** In a condition, the stream is interpreted as `true` if it is still good, and as false on failure.

When reading characters using `cin.get()`, input fails **only if there are no characters left** in the stream. The effect of the **basic data loop** is to execute the body of the `while` loop once for **each** character until the stream reaches what is known as **end of the file**.

For **output streams**, the `put()` member function takes a `char` value as its argument and writes that character to the stream.

More on Streams

When writing a function which processes an input or output, the stream parameters must always be **passed by reference**.

Here, for example, is a function that copies input to output.

```
void streamCopy(istream& in, ostream& out)
{
    char ch;
    while (in.get(ch)) { out.put(ch); }
}
```

We could rewrite ***filter*** by **calling this function**, like this:

```
int main()
{
    streamCopy(cin, cout);
}
```

Other I/O Functions

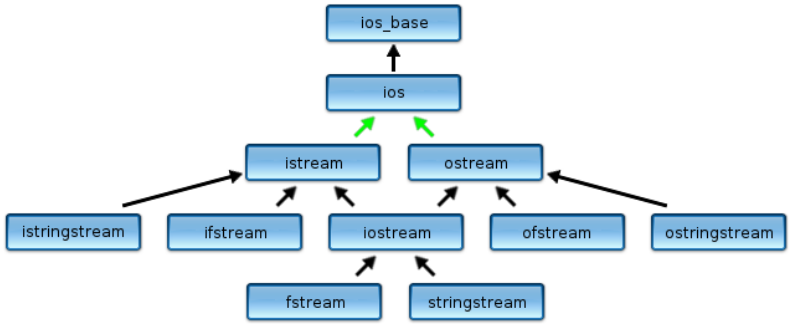
When reading individual characters, you'll sometimes find that you have **read more than you need**. There are several ways to solve this problem in C++.

1. `in.unget()` returns the last read character **to the input stream**.
2. `in.putback(ch)` allows you to put back a **different character**.
3. `in.peek()` looks at the next character in the stream, but doesn't remove it from the stream.

The C++ library guarantees that it you can push back push one character. You are not able to read several characters ahead and then push them all back. Fortunately, being able to push back one character is sufficient in the vast majority of cases.

The Stream Classes

The C++ standard library stream headers contain several different classes that form a **class hierarchy**, designed using the object-oriented facility known as **inheritance**.



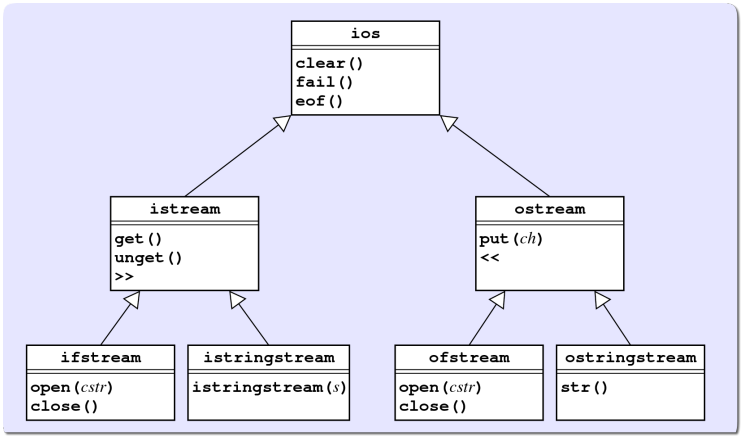
Note **headers**, not header. Until now, have one stream header: `<iostream>`. To read and write to files (instead of the standard streams, we'll use the **file stream** classes—`ifstream` and `ofstream`—found in the `<fstream>` header. The name `ifstream` stands for **input-file-stream,** while the name `ofstream` stands for **output-file-stream**.

In the diagram above, each class is a **derived class** (or **subclass**), of the class above it. Thus, `istream` and `ostream` are both **derived from** `ios`, and are **specialized** kinds of `ios` objects. In the opposite direction, `ios` is a **base class** (or **superclass**) of both `istream` and `ostream`. Similarly, `ifstream` is derived from `istream` and `ostream` is the base class of `ofstream`.

This relationship—between base and derived classes—is conveyed by the words **is a**. Every `ifstream` object **is a** `istream and`, by continuing up the hierarchy, an `ios`. This means that characteristics of any class are **inherited** by its derived classes.

UML Diagrams

Simple diagrams that show the relationships among classes are useful, but often we want to expand them to **include the member functions** exposed at each level. This diagram is a standard way of displaying a class hierarchy called the **Unified Modeling Language**, or **UML.** In UML, each class appears as a rectangular box whose upper portion contains the name of the class.



In a UML class diagram, the **public member functions** of the class appear in the lower portion. In UML, derived classes use open arrowheads to **point to** their base classes.

UML diagrams make it easy to determine which **inherited member functions** are available to each of the classes in the diagram. Because each class inherits all of the members of **every class** in its ancestor chain, an object of a particular class can call **any member function** defined in any of those classes.

For example, the diagram above shows that **any** `ifstream` object can call these member functions:

- The `open()` and `close()` functions from the `ifstream` class itself
- The `get()` and `unget()` member functions, as well as the `>>` operator from the `istream` class
- The `clear()`, `fail()`, and `eof()` functions from the `ios` class

Easy File I/O

File processing in C++ is fairly straightforward:

1. **Declare a stream variable to refer to the file**. Here's an example with both an input file stream and an output file stream.

```
1 ifstream infile;
2 ofstream outfile;
```

2. **Open the file.** To **establish an association** between that variable and an actual physical file on disk you need to **open the file** calling `open()`.

```
infile.open("myfile.txt");
```

   Alternatively, you can use perform **both steps at once** using the **stream constructors**. Here's an example:

```
ifstream infile("myfile.txt");
```

   If the **file is missing** the stream will **fail to open**; you can check for that by calling the member function `fail()`. There will be **no other error messages**:

```
1 ifstream infile("myfile.txt";
2 if (infile.fail()) { /* handle error */ }
```

3. **Transfer the data.** Read and write data using these techniques:

   - Read or write character by character using **unformatted I/O**.
   - Process the file **line by line**, using **line-oriented I/O**.
   - Read and write **formatted data**, mixing numeric data with strings and other data types. This is known as **token-based file I/O**.