Chapter

11

# Applying Recursion

Now that you've seen several examples of recursion, let's apply recursion to a few problems, similar to those on a programming exam. To help you, here is a checklist that will help you identify the most common sources of error.

## Check the Simple (Base) Cases

Does your recursive implementation begin by checking for simple base-cases? First check to see if the problem is so simple that no recursion is necessary. Recursive functions begin with the keyword `if`; if yours doesn't, look carefully.

## Have You Solved the Base Cases Correctly?

A surprising number of bugs in arise from **incorrect solutions to the simple base-cases**. If the simple cases are wrong, the rest of the function will inherit the same mistake. If you had defined `fact(0)` as `0` instead of `1`, **any** argument would end up returning `0`.

## Does Decomposition Make it Simpler?

Does your recursive decomposition **make the problem simpler**? Problems have to get simpler as you go along; the problem **must get smaller** as it proceeds. If the problem does not get simpler, you have the recursive analogue of the infinite loop, which is called nonterminating or infinite recursion.

## Have You Covered All Possibilities?

Does the simplification process always reach the base case, or **have you left out some of the possibilities**? A common error is forgetting one of the base-cases. You need to check the empty string in the `isPalindrome()` function, as well as the single-character string. Since the function reduces the size of the string by two each time, only having the one-character base case, would mean some strings would fail.

## Are Sub-problems Identical in Form?

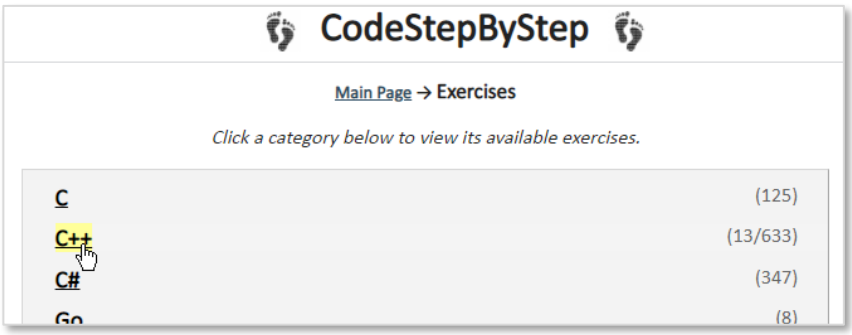
Are the recursive sub-problems **truly identical in form to the original?** It is essential that the sub-problems be of the same form. If the recursive calls change the nature of the problem then the entire process can break down.

### *Are You a Believer?*

When you apply the recursive leap of faith, do the solutions to the recursive sub-problems **provide a complete solution to the original problem**? Work through all the steps in the current function call, but assume that every recursive call returns the correct answer. If this process yields the right solution, your program should work.

## Try It Yourself

On the Canvas course home page, you'll find a link to **Code Step-by-Step**, a Web site providing practice problems in several different programming languages, including C++. Go ahead and create your own account, and then let's walk through a problem.




## CollapseSequences

Click the C++ link as shown in the screenshot above, and then find the **recursion** section. Here are the instructions for the first problem, `collapseSequences`.

> *Write a **recursive function** named `collapseSequences` that accepts a `string s` and `char c` as parameters and returns a new `string` that is the same as `s` but with any sequences of **consecutive occurrences** of `c` compressed into a single occurrence of `c`. For example, if we collapse sequences of character `'a'` in the string `"aabaaccaaaaada"`, you get `"abaccada"`.*
>
> *Your function is **case-sensitive**; if the character `c` is, for example, a lowercase `'f'`, your function should not collapse sequences of uppercase `'F'` characters. In other words, you do not need to write code to handle case issues in this problem.*

The following table shows two examples and their expected return values:

| Call | Returns |
|---|---|
| `collapseSequences("aabaaccaaaaaada", 'a')` | `"abaccada"` |
| `collapseSequences("mississssipppi", 's')` | `"misisipppi"` |

## *Solve the Problem*

Let's walk through the steps listed in the checklist. Normally, of course, you'll compress these steps together in an intuitive way. However, if you have difficulty with solving a recursive problem, going back and checking them individually is a good debugging tool.

1. **What is the simplest base case?** In other words, what values for s require **no compression**? Well, obviously, if we don't have any characters, there can be no compression. Similarly, if we have only a single character, there can be no compression. The **simplest thing that can work, without recursion** is:

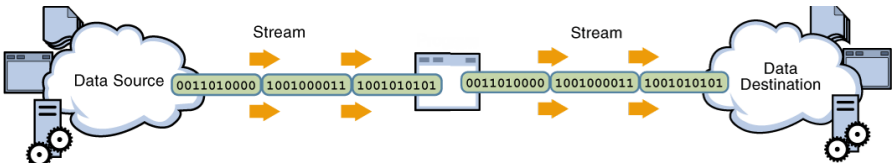
```
string collapseSequences(const string& s, char c)
{
    if (s.size() < 2) return s; // base case
}
```

2. **Have we handled all of the base cases?** The empty string returns "", and a single character returns that character. It doesn't matter if the character matches the parameter `c`, since a single-character cannot be a sequence. Those should be all of the base cases. If we have two charcters, we may have a sequence "cc" that requires compression.
3. **Does decomposition make it simpler?** Or, in other words, how can we solve a simpler version of the problem? Or, how can we approach the base case? How, by **passing a smaller** `string` each time we call the function recursively.
4. **Have you covered all possibilities?** Since we have two characters, they **may** both match `c`, or they **may not**. If they **do not**, then we need to **include** the first character in the returned `string`, and pass a shortened `string` to our function. **If they do** both match `c`, then we need to **ignore** the first character, and pass only our shortened `string` to the function. This should **cover all possibilities.**
5. **Are the subproblems identical in form?** We do a different action in each case; in one we include the first character, and in other we do not, but the **form** of the problem **is the same in both cases.**
6. **Are you a believer?** Walk through a solution using the simplest recursive cases. If it works for those, then **it must work for all of them**. Let's assume that `c` is the character `'v'` and `s` is `"vv"`

a. The two characters match `c`, so the first `v` is **ignored** and the function is **called again** with the `string` `"v"`.
b. On the second recursive call, the base case returns `"v"`
c. Thus, the sequence `"vv"` is compressed to `"v"` **so it works**.

# Streams & Redirection

**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**. Let's look at an operating facility named redirection.

## Standard Streams

Let's start with a question. When you run a program, **where** do the input bytes are **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?

When you run a program, the **operating system** automatically opens three streams:

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

In a C++ program, the built-in streams are used to initialize the `cin`, `cout` and `cerr` I/O objects. (In Java, they initialize the `System.in`, `System.out`, and `System.err` objects).

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**.

## Input Redirection

Right-click the "running man" and choose "Open in a new tab" to **open a sandbox** which already contains a few files. Let's look at a few built-in Unix filters.

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 **CONTROL+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 command is named **type**.
- The filter **stops** reading when it reaches **end-of-file**. In Unix, you simulate that by typing **CONTROL+D** from the terminal. In Windows, it is **CONTROL+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**.

To see how input redirection works, open the file named **input.txt** by clicking its tab. Then, type this command in the shell:

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

The **input redirection symbol** **<** asks the operating system to read **input.txt** as the **standard input**. 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.

# Output Redirection

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

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

No output will appear at all on your screen; instead, the file **output.txt** will be created and all the output will 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 contents.

See the file by using **cat** again, this time with **output.txt**, like this:

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

Instead of erasing existing data, you can **append** like this:

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

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

# Output and Error Streams

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**.

**Redirect standard error** 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
```

## The Special File /dev/null

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

```
$ 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 and 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 as input **to 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
$ cat sorted.txt
```

You can save it to a file using output redirection:

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

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.

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

Here is a 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
```

# Filters

**Filters may change, use, or learn about the characters flowing through your** program. Two general kinds of filter programs are **process** filters and **state** filters. A process filter **does something to** the characters it encounters, while a state filter **learns something about** the stream by examining characters.

## Process Filters

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 discard would actually be simpler). That's what the **`cat`** command is.


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 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 newline)
- 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.

# Unformatted I/O

Now, let's look at **writing our own filters**. We'll start with the simplest possible version of ***cat.*** Remember, 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)

Let's start by looking at **raw, unformatted input**. Build and run `filter.cpp` in the sandbox, like this:

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

Note, 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.

## The Data Loop

C++ input streams read a single character using the **member function** `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 and process one data item
```

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

```
char ch;
while (cin.get(ch))
{
    // do something with ch
}
```

## Input Stream Conditions

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 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. The stream is interpreted as `true` on success and as `false` on failure.

When reading characters, 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.

## Unformatted Output

For output streams, the `put()` method takes a `char` value as its argument and writes that character to the stream. A typical call to `put()` therefore looks like this:

```
cout.put(ch);
```

# Stream Parameters

When writing a function which processes an input or output stream, write it to use the class `istream` for input, and the class `ostream` for output. Stream parameters must always be passed by reference.

Here 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.

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.

# Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.