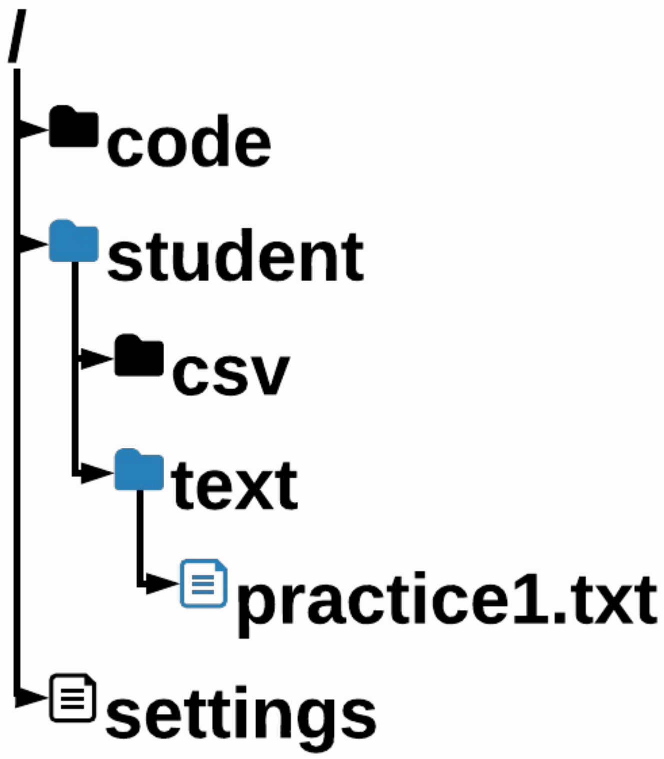
Learning Objectives: Reading

- **Demonstrate how to open a file using `ifstream`**
- **Use the `get()`, `getline()`, and `rdbuf()` functions to read a string stream**
- **Iterate through a file using a `while` loop**
- **Define the term “token”**
- **Tokenize a string read from a file using a delimiter**
- **Ignore characters in a file with the `ignore()` function**

File Basics

File Basics

This module is all about working with files on a computer. The first step is to locate the desired file. That means being able to navigate the file system. The file we are going to use is called `practice1.txt`. It is located in the `text` folder, which is inside the folder called `student`. So the path to the file is: `student/text/practice1.txt`.



File Path

Use a string to represent the file path. This string will be passed to direct the system to open a file.

```
string path = "student/text/practice1.txt";
```

Opening and Closing Files

One of the most common ways to open, close, read, and write files is to use the `ifstream` or `ofstream` data type. The difference between `ifstream` and `ofstream` is that `ifstream` is used to read data from files whereas `ofstream` is used to write to files. Both of these data types can be accessed and utilized by including `#include <fstream>` in the header of your code file. Let's try to open `practice1.txt` as specified from above using `ifstream`.

```
string path = "student/text/practice1.txt";

ifstream file;
file.open(path);
```

You'll see that `Command was successfully executed.` is returned, which isn't very helpful. To know if the file was opened successfully, you can set up conditions like so:

```
string path = "student/text/practice1.txt";

ifstream file;
file.open(path);
if (file.is_open()) {
  cout << "File successfully opened." << endl;
}
else if (!file.is_open()) {
  cout << "File failed to open." << endl;
}
```

challenge

What happens if you:

- Change `if (file.is_open())` to `if (file)`? → File successfully opened
- Change `else if (!file.is_open())` to `else if (!file)`? → File successfully opened
- Change the string `path` to `"student/text/practice2.txt"`? → File failed to open

Before your program terminates, it is a best practice to close the file. When a file is opened, it takes up memory that will not free up until the file is properly closed.

```
string path = "student/text/practice1.txt"; // setting file path

ifstream file; //create a read-only data stream
file.open(path); //open the file as a stream
if (file) { //check if file exists
  cout << "File successfully opened." << endl;
}
else if (!file) { //check if file does not exist
  cout << "File failed to open." << endl;
}
file.close(); //close the file
```

File Error Handling

Back in the "User-Defined Functions" module, you were introduced to the `try`, `throw`, and `catch` blocks which are used to handle runtime errors that occur as the program runs. You can apply the same concept to catch errors when opening files which is the preferred way to handle these errors.

```
string path = "student/text/practice1.txt";

try { //try these actions
  ifstream file;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open."); //throw error
  }
  file.close();
  cerr << "File successfully opened and closed." << endl;
}

catch (exception& e) { //catch error
  cerr << e.what() << endl;
}
```

challenge

What happens if you:

- Change the string `path` to `"student/text/practice2.txt"`? → File failed to open info

What is `cerr`?

When printing error messages, `cerr` is preferred over `cout`. `cerr` is not bufferred, which means it is not stored in memory to be printed later on. It just gets printed immediately. Therefore, as a rule of thumb, important data and variables should be printed with `cout` while error messages should be printed with `cerr`.

Reading a File

Reading a File

Let's start reading from a file that contains some text. First set the `string path` to `student/text/readpractice.txt`. Then open the file and handle any opening errors. To read from a file, use the `getline()` function. The `getline()` has at least two parameters; the first is the input stream to read from and the second is a string to store what is read.

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  string read; //create string to store what is read into
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  getline(file, read);
  cout << read;
  file.close();
  cerr << "File successfully opened and closed." << endl;
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

important

IMPORTANT

You’ll notice from above that the system printed `File successfully opened and closed.` first and then `C++ was created by Bjarne Stroustrup.` was printed. This is due to `cerr` being unbuffered, which means it gets printed immediately. `cout` is buffered, so it will store its content into memory before printing, which explains the delay. This doesn’t mean that `cerr` always gets printed first, it just depends on what on quickly `cout` stores its content.

To see the content of the file `readpractice.txt`, click this link here: readpractice.txt

If you take a look at the content of the file, you’ll see that it has more text than just `C++ was created by Bjarne Stroustrup.`. The reason why only one line of text was printed is because `getline()` only reads up until a `newline` character is reached. To continue to read more lines of text, wrap `getline()` inside a `while` loop.

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  string read; //create string to store what is read into
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (getline(file, read)) {
    cout << read;
  }
  file.close();
  cerr << "File successfully opened and closed." << endl;
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

challenge

What happens if you:

- Change `cout << read;` in the code to `cout << read << endl;`?
- Change the entire code to:

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  char ch; //create string to store what is read into
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (file.get(ch)) {
    cout << ch;
  }
  file.close();
  cerr << "File successfully opened and closed." << endl;
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

- Change `cout << ch;` in the new code to `cout << ch << endl;`?

The `get()` function in the new code works similarly to the `getline()` function. However, `get()` reads character by character instead of by strings.

Additionally, did you notice that `File successfully opened and closed.` was printed at two different locations before and after `<< endl` was added? This all depends on how quickly `cerr` and `cout` work behind the scenes. To avoid inconsistencies in printing, we'll avoid printing the success message moving forward.

Reading a Buffer

The rdbuf() Function

The `getline()` and `get()` functions require a variable to store its content. If all you want to do is to read a file however, then the variable becomes useful and simply takes up extra memory. When a file is opened, its content is buffered or stored in memory until it gets closed. During this buffering stage, you can use the function `rdbuf()` to read the content of the file without having to create any variables.

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  file.open(path); //content of file goes into memory buffer
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  cout << file.rdbuf(); //read the buffered content
  file.close();
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

challenge

What happens if you:

- Switch the line `cout << file.rdbuf();` with `file.close()`?

If the file is closed, the buffer gets flushed to clear the memory that was used. So after a file is closed, you will not be able to read the buffered content anymore.

Delimiters

Delimiters

Delimiters are a predefined character that separates one piece of information from another. Some common delimiters involve white spaces (`' '`) and commas (`','`). When using `getline()` previously, we only made use of two of its parameters. It actually has a third parameter which is reserved for a delimiter. By default, this delimiter is the newline character (`'\n'`). Thus the parameters for `getline(x, y, z)` are:
* The stream source (x).
* The string variable to store what is read (y).
* The delimiter to separate the content of the stream source (z).

Let's specify the `getline()` function to use a comma as a delimiter `,`.

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  getline(file, read, ','); //specify comma as delimiter
  cout << read;
  file.close();
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

Click to see file's content: readpractice.txt

When a delimiter is applied, the system will read only up to that delimiter. This is why you only see content up through the first comma occurrence. If you want to continue reading further and get to the other comma occurrences, you can put the command inside a `while` loop.

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (getline(file, read, ',')) { //specify comma as
        delimiter
    cout << read;
  }
  file.close();
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

Click to see file's content: readpractice.txt

The code above continues to read the stream and separates the content via the delimiter, `,`. This is why there are no commas present in the output. However, it doesn't look very clear that the stream has been separated since the system continues to print the remainder of the stream. You can add `<< endl` to the output so that the system will print a newline after each delimiter is reached so that you can see clearer where the content is separated.

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (getline(file, read, ',')) { //specify comma as
        delimiter
    cout << read << endl;
  }
  file.close();
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

Click to see file's content: readpractice.txt

challenge

What happens if you:

- Change the delimiter argument from `','` to a white space `' '`?
- Change the delimiter again to an `'a'`?
- Change the delimiter again to a newline `'\n'`?

Click to see file's content: readpractice.txt

Tokens

Tokens

When we apply a delimiter to break up a stream or string, the resulting separated strings are sometimes referred to as **tokens**. Tokens are useful if you want to use their data for further analysis later on. For example, you can store each token as an element in a vector in which you can extract further information from later on. What do you think the code below does?

The following file contains the first and last names of 5 individuals: names.txt

```
string path = "student/text/names.txt";
vector<string> names;
string last_name = "Smith";
int count = 0;

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (getline(file, read, ' ')) {
    names.push_back(read);
  }
  file.close();
  for (int i = 0; i < names.size(); i++) {
    if (i % 2 == 1) {
      if (names.at(i) == last_name) {
        count++;
      }
    }
  }
  cout << "There are " << count << " people whose last name is
        Smith." << endl;
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

The code above enables you to do several things:
1. Break the stream into several tokens separated by a white space.
2. Store the tokens into a vector.
3. Iterate through the vector to count how many people have the last name “Smith”.
4. Print the resulting count.

challenge

What happens if you:

- Try to search for a first name such as `"Jackie"` instead?
  - ▼ **Sample Solution**

```
string path = "student/text/names.txt";
vector<string> names;
string first_name = "Jackie"; //change variable to first
        name
int count = 0;

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (getline(file, read, ' ')) {
    names.push_back(read);
  }
  file.close();
  for (int i = 0; i < names.size(); i++) {
    if (i % 2 == 0) { //start checking index 0 and then
        every other index
      if (names.at(i) == first_name) { //first name found
        count++;
      }
    }
  }
  cout << "There are " << count << " people whose first
        name is Jackie." << endl;
} //change the print statement as needed

catch (exception& e) {
  cerr << e.what() << endl;
}
```

Applying Another Delimiter

If you need to further break down your tokens, you can use a nested loop to iterate through those tokens to break them down even further. For example, if your list of names was organized like this:

```
Jason Seymore
Jackie Simmons
Jennifer Small
Jane Smith
John Smith
```

Then using the code above will cause issues since it only takes a white space as a delimiter, not a newline. To include both the newline and white space as delimiters, you can use a `stringstream` data type (`#include <sstream>`) to create another string stream off of the string variable `read`. The first `getline()` function will separate the stream using a newline as the delimiter and the second `getline()` will use a white space as a delimiter.

```
string path = "student/text/names2.txt";
vector<string> names;
string last_name = "Smith";
int count = 0;

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (getline(file, read)) { //newline delimiter
    stringstream ss(read); //create a string stream of read
    while (getline(ss, read, ' ')) { //white space delimiter
      names.push_back(read);
    }
  }
  file.close();
  for (int i = 0; i < names.size(); i++) {
    if (i % 2 == 1) {
      if (names.at(i) == last_name) {
        count++;
      }
    }
  }
  cout << "There are " << count << " people whose last name is
        Smith." << endl;
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

Click to see files' content: names2.txt and names.txt

challenge

What happens if you:

- Change the string path from `"student/text/names2.txt"` to `"student/text/names.txt"`?

Notice how you get the same result regardless of how your names are organized in the text file.

Ignore Function

The Ignore Function

The `ignore` function takes an integer as a parameter, and causes C++ to go to a specific character in the text file. The integer is the index for the text file. So `ignore(0)` is the first character of the file, `ignore(1)` is the second character, etc. The code below prints out the entire text file.

Click to see file's content : readpractice.txt

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  while (getline(file, read)) {
    cout << read << endl;
  }
  file.close();
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

Now compare the output above with the output from the code below. C++ will ignore the first 29 characters and start reading only from the character at position `30`.

```
string path = "student/text/readpractice.txt";

try {
  ifstream file;
  string read;
  file.open(path);
  if (!file) {
    throw runtime_error("File failed to open.");
  }
  file.ignore(30); //ignore all chars before index 30
  while (getline(file, read)) {
    cout << read << endl;
  }
  file.close();
}

catch (exception& e) {
  cerr << e.what() << endl;
}
```

challenge

Try these variations:

- Change the ignore argument to 40: `file.ignore(40);`
- Change the ignore argument to 400: `file.ignore(400);`

▼ **Why do I see `Command was successfully executed.`?**

The text file only has 242 characters total, which include white spaces and newlines. Ignoring all characters before index `400` is like ignoring the entire file. Nothing gets read or printed which is why the system returned `Command was successfully executed.`.