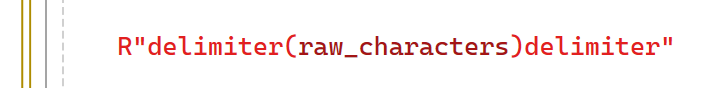
## 11. File Input & Output

# Raw String Literals

Raw string literals are a feature introduced in C++11 that allow you to define string literals without the need for escaping special characters, such as backslashes (`\`). This is particularly useful for representing strings that contain regular expression patterns, file paths, JSON data, HTML markup, or any other content that includes a lot of escape characters.

In C++, a raw string literal is denoted by surrounding the string with `R"(` and `)"`, followed by an optional delimiter. Here's the syntax:





* `R"(`: The beginning of the raw string literal.
* `delimiter`: An optional sequence of characters that delimits the raw string. This delimiter can be any sequence of characters, but it cannot contain the sequence `)`, which is used to terminate the raw string literal.
* `raw\_characters`: The content of the raw string literal, which can include any characters, including newlines and escape sequences.

**For example:**

A screen shot of a computer code

Description automatically generated

* In this example, the raw string literal `R"(`...`)"` is used to define an HTML snippet, including multiple lines and indentation, without needing to escape any characters.
* Raw string literals simplify the representation of complex strings and improve code readability by eliminating the need for excessive escaping of special characters. However, it's important to choose an appropriate delimiter that does not appear in the content of the string.

# Filesystem Library

A screen shot of a computer code

Description automatically generated

* We include the <experimental/filesystem> header to access the file system library.
* We alias the std::experimental::filesystem namespace as fs for brevity.
* We create a path object representing a file path, using a raw string literal.
* We use the has\_filename() function to check if the path contains a file name, and filename() to retrieve it.
* We iterate over the components of the path using a range-based for loop.
* We create a directory\_iterator object to iterate over the contents of a directory, specifying the directory path.
* We iterate over the directory contents, printing out the file names.

# Basic File I/O

A white background with black text

Description automatically generated

A computer screen shot of a diagram

Description automatically generated

A diagram of a class type of software

Description automatically generated

A diagram of a class

Description automatically generated

A diagram of a company

Description automatically generated

A screenshot of a computer

Description automatically generated

# A summary of the main points you mentioned:

1. **Classes for File I/O:**

* `**ofstream**`: Used for **writing** to an output stream (output file).
* `**ifstream**`: Used for **reading** from an input stream (input file).
* `fstream`: Used for both reading from and writing to a stream (input/output file).
* All these classes are defined in the `<fstream>` header.

1. **Modes for File Operations:**

* When creating instances of these classes for file I/O, you can specify modes that determine how the file is opened and operated.
* Modes include options like reading **(`std::ios::in`),** writing **(`std::ios::out`),** appending **(`std::ios::app`),** binary mode **(`std::ios::binary`),** etc.

1. **Checking Stream Status:**

* To check if a stream has been opened successfully, you can use the **`is\_open()`** function, which returns a boolean indicating whether the stream is open or not.

1. **Class Hierarchy:**

* The hierarchy of stream classes starts with `basic\_ios`, which is the base class.
* **`ios`** is a derived class **of `basic\_ios`, and `ios\_base`** contains common state and settings for all stream classes.
* Derived from **`ios`** are `istream` (for input) and `ostream` (for output), used for console input-output.
* For file I/O, C++ provides `ifstream` (input file) and `ofstream` (output file).
* `fstream` can be used for both reading from and writing to a file.
* Additional classes like `**istringstream**` and `**ostringstream**` are used for reading from and writing to a buffer, while `**stringstream**` allows reading from and writing to a string buffer simultaneously.

# File Open Modes

1. **app (Append):**

* Indicates that when a file is opened for writing, it seeks to the end of the file before each write operation. This allows new data to be appended to the existing content.

1. **binary:**

* Used to open files in binary mode. In binary mode, data is read or written without any translation or formatting, which is useful for handling non-text files or ensuring platform-independent file I/O.

1. **in:**

* Used for opening a file in read mode. It is the default mode for the constructor of `ifstream` class, which is used for reading from an input file.

1. **out:**

* Used for opening a file in write mode. It is the default mode for the constructor of `ofstream` class, which is used for writing to an output file.

1. **trunc (Truncate):**

* Indicates that the contents of the file should be discarded before opening. If the file already exists, its contents are deleted. This mode is commonly used when you want to overwrite an existing file with new content.

1. **ate (At End):**

* Used to seek to the end of the file immediately after opening. This is useful when you want to append data to an existing file without overwriting its contents.
* These modes can also be combined using the bitwise OR (`|`) operator. For example, combining `std::in` and `std::out` allows reading and writing to a file simultaneously. Understanding and utilizing these modes effectively is essential for performing file I/O operations in C++ with the desired behavior and performance.



1. **Header File Inclusion**: Begin by including the `<fstream>` header file, which provides classes and functions for file I/O operations.
2. **Write Function**: Create a function called `Write()` to write data to a file. Use an instance of `ofstream` to write to the file. Optionally, specify the mode (default is `std::out`). Use the insertion operator **(`<<`)** to write data to the file. Close the file after writing.
3. **Read Function:** Create a function called `Read()` to read data from the file. Use an instance of `ifstream` to read from the file. Specify the filename and create a string object to store the read data. Use the **`getline()`** function to read an entire line from the file. Optionally, read other data types (e.g., integers) using extraction operators **(`>>`).** Close the file after reading.
4. **Usage of Constructors and open() Function:** Instead of using constructors to open files, you can use the `open()` function available in both `ofstream` and `ifstream` classes. This function accepts the filename as an argument and allows you to open the file.
5. **Calling Functions in main():** Call the `Write()` and `Read()` functions in the `main()` function to perform file I/O operations.

# File I/O Error Handling

* To test whether a file has been opened successfully or not, you can use the `is\_open()` function provided by the stream classes in C++. Here's how you can incorporate it into your file I/O code:

A screenshot of a computer program

Description automatically generated

**In this example:**

* We attempt to open a file using `ifstream` and the `open()` function.
* We immediately check if the file is opened successfully by calling the `is\_open()` function on the file stream.
* If the file is opened successfully, we perform the necessary reading operations inside the `if` block.
* If the file fails to open (for example, due to a wrong filename), we output a failure message.
* Finally, regardless of whether the file is opened successfully or not, we close the file after performing the read operation.

This approach allows you to handle cases where the file opening operation may fail due to incorrect filenames or other reasons.

A screenshot of a computer

Description automatically generated

In C++, the stream classes have several stream state flags that provide information about the state of the stream. These flags are automatically set when the state of the stream changes. Here's an overview of the stream state flags and how they can be used:

1. **goodbit:** Indicates that there is no error with the stream. The `good()` function returns true when this flag is set.
2. **badbit:** Indicates that the stream has encountered a critical error from which it cannot recover. This could be due to a disk I/O error or corrupted media. The `bad()` function returns true when this flag is set.
3. **failbit:** Indicates that an input-output operation has failed, but the stream can still recover. For example, this flag is set if a file could not be opened or if characters cannot be read from the file. The `fail()` function returns true when this flag is set.
4. **eofbit:** Indicates that the end of the file has been reached during an input operation. The `eof()` function returns true when this flag is set.

* To check the state of these flags, you can use the respective member functions (`good()`, `bad()`, `fail()`, `eof()`) of the stream classes. For example:

A screenshot of a computer code

Description automatically generated

* Additionally, you can clear the stream state flags using the `clear()` function. To set a specific state flag, you can use the `setstate()` function. For example:

A close up of a text

Description automatically generated

# File I/O Copy Utility

A screen shot of a computer program

Description automatically generated

**Here's a concise overview of the text file copying utility:**

1. Include the necessary header file for the file system library.
2. Open the `std::filesystem` namespace.
3. Create a path to the source file using the `current\_path()` function.
4. Use the compound assignment operator to append the file name to the path.
5. Create a destination path by adding a new file name, such as "Copy.cpp".
6. Open the source file and check if it is successfully opened.
7. If the source file is opened, create the destination file “dest”.
8. Read the source file line by line using `std::getline()` in a while loop.
9. Check if the end of the file is reached using the `eof()` function.
10. Write each line to the destination file.
11. Close both the source and destination files.

This utility currently supports only the copying of text files, not binary files. However, future modifications may extend its functionality to support additional file types.

# **The current\_path()**

* Function in C++'s filesystem library is used to obtain the current working directory of the program. When called, it returns a `std::filesystem::path` object representing the path to the current working directory.
* For example, if your program is located in the directory `/home/user/project/` and you call `current\_path()` within the program, it will return a `std::filesystem::path` object representing the path `/home/user/project/`.
* You can then use this path object to perform various file system operations relative to the current working directory, such as creating or opening files, navigating directories, or constructing paths for file operations.

**The line of code `source /= "Source.cpp";` appends the file name `"Source.cpp"` to the `source` path. Here's what each part of the line does:**

* `source`: This is a `std::filesystem::path` object representing the path to the current working directory obtained using `current\_path()`.
* `/=`: This is the compound assignment operator for the `std::filesystem::path` class. It appends the specified path component to the existing path.
* `"Source.cpp"`: This is the file name that we want to append to the `source` path.

So, in essence, after this line of code executes, the `source` path will represent the path to the current working directory with the file name `"Source.cpp"` appended to it. This allows us to reference the source file within the current directory for file operations.

**.eof():** This is a member function of the input stream that checks whether the end-of-file has been reached. It returns true if the EOF flag is set on the stream, indicating that no more characters can be read from it

**The line of code “ output << line << std::endl;”**

* Get the line from the file “input” and put it in the string “line”, then put the string in the file “output” the add a new line

A screenshot of a computer program

Description automatically generated

# File I/O Character IO & Seeking

A screenshot of a computer program

Description automatically generated

A close-up of a computer code

Description automatically generated

**This code creates an std::ofstream object named out and attempts to open a file named "text.txt" for writing. Here's what each part of the code does:**

* Std::ofstream out("text.txt");: This line declares an std::ofstream object named out and initializes it with the file name "text.txt". This creates an output file stream object associated with the specified file.
* If (!Out) { ... }: This if statement checks if the file stream object out is in a failed state. If the file failed to open for writing (due to reasons such as insufficient permissions or non-existent file), the condition !Out evaluates to true, indicating that the file stream object is in a failed state.

In the context of the code provided, the file "text.txt" will be created in the current working directory where the program is executed. If you haven't specified a specific directory path for the program to execute in, it will **typically be the directory from which you executed the program** or the default directory specified by your development environment or operating system.

**For example,** if you run the program from the command line without specifying a directory, the file "text.txt" will be created in the directory where the executable file of your program is located. Similarly, if you run the program from an integrated development environment (IDE) like Visual Studio or Code::Blocks, the file will be created in the project directory or the directory specified by the IDE for program execution

If you want to create the file "text.txt" in a specific directory other than the current working directory, you can specify the full path to that directory when opening the file. Here's how you can modify the code to achieve this:

A screenshot of a computer code

Description automatically generated

Replace `"/path/to/your/directory/text.txt"` with the full path where you want to create the file. This can be an absolute path (e.g., "/home/user/documents/text.txt") or a relative path (e.g., "C:/Users/User/Documents/text.txt" on Windows).

A computer screen shot of text

Description automatically generated

**This code writes each character of the string "Mina Magdy Aziz" to the output stream `out`. Here's what each part of the code does**:

* `std::string message{ "Mina Magdy Aziz" };`: This line declares a `std::string` object named `message` and initializes it with the string "Mina Magdy Aziz".
* `for (char ch : message) { ... }`: This is **a range-based for loop** that iterates over each character (`char`) in the string `message`.
* Inside the loop: For each character `ch` in the string `message`, the `out.put(ch)` statement writes the character to the output stream `out` using the `put()` member function of `std::ofstream`. This effectively writes each character of the string to the output file one by one.

Overall, this loop iterates over each character in the string "Mina Magdy Aziz" and writes them to the output stream `out`, which should be the output file opened earlier.

A computer code with red text

Description automatically generated

**This code attempts to open the file "text.txt" for reading using a std::ifstream object named input. Here's what each part of the code does:**

* Std::ifstream input{ "text.txt" };: This line declares a std::ifstream object named input and initializes it with the file name "text.txt". This creates an input file stream object associated with the specified file for reading.
* If (!Input) { ... }: This if statement checks if the file stream object input is in a failed state. If the file "text.txt" is not found or cannot be opened for reading, the condition !Input evaluates to true, indicating that the file stream object is in a failed state.

**For Specific path:**

A close up of text

Description automatically generated

# If we want to read or write in a specific position

A screenshot of a computer program

Description automatically generated

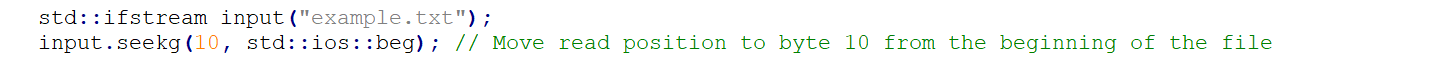
1. **`seekg()` and `seekp()`:**

* `seekg()` is used to set the read position within an input file stream (`std::ifstream`).
* `seekp()` is used to set the write position within an output file stream (`std::ofstream` or `std::fstream`).

**Both functions take two parameters or one parameter:**

* The first parameter is the position to seek to within the file, typically specified as an offset from a reference point.
* The second parameter specifies the reference point from where the position is calculated. This parameter is optional and defaults to `std::ios::beg` (beginning of the file).

**Example usage:**



1. **Reference points (`std::ios::beg`, `std::ios::cur`, `std::ios::end`):**

* **“std::ios::beg”** (beginning): Specifies that the position is relative to the beginning of the file.
* **“std::ios::cur”** (current): Specifies that the position is relative to the current read or write position.
* **“std::ios::end”** (end): Specifies that the position is relative to the end of the file.
* These reference points are used as the second parameter in `seekg()` and `seekp()` functions to specify the position relative to a particular point within the file.

1. **Overloads of `seekg()` and `seekp()`:**

* These functions have multiple overloads that provide flexibility in setting the file position based on different reference points and seeking modes. Overloads include versions that take streamoff or streampos parameters, and versions that seek relative to the current position or the end of the file.

1. **`tellg()` and `tellp()`:**

* `tellg()` returns the current read position within an input file stream.
* `tellp()` returns the current write position within an output file stream.
* These functions return a value of type `std::streampos`, which represents the current file position indicator within the stream.
* These functions and reference points are essential for managing file positions and enabling random access operations within input and output file streams.

# Fstream

A computer code with red and white text

Description automatically generated

1. Include necessary header files (`<iostream>`, `<fstream>`, `<string>`). These headers provide functionality for input and output operations, file handling, and string manipulation, respectively.
2. Open the file using `std::fstream`. The file is opened in both input and output modes.
3. Check if the file exists. This is done to verify if the file exists before attempting to open it. If the file doesn't exist, it needs to be created.
4. Check if the file is open. This ensures that the file has been successfully opened and is ready for input and output operations.
5. Write a message to the file. This writes a message to the file using the output operator (`<<`). The message is appended to the end of the file.
6. Seek the get pointer to the beginning of the file. This moves the get pointer to the beginning of the file so that subsequent read operations start from the beginning.
7. Read and display the content of the file. This reads the contents of the file line by line using `std::getline()` and displays each line on the console.
8. Close the file. This closes the file once all operations are completed. It is important to close the file to release system resources and ensure data integrity.

# File IO Binary IO

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

Another example of a struct



# Relative function

* **In the provided code, `relative(entry.path(), sourcedir)` is a function call that calculates the relative path of a file with respect to a given directory. Here's a detailed explanation of this function call:**
* `entry.path()`: This returns the absolute path of the file represented by the directory iterator `entry`. It represents the full path of the current file being iterated over.
* `sourcedir`: This is the base directory from which the relative path is calculated. It represents the base directory where the file copy operation started.
* `fs::relative()`: This is a function provided by the `std::filesystem` library in C++17 and later. It calculates the relative path of one path with respect to another path. In this case, it calculates the relative path of the file (given by `entry.path()`) with respect to the source directory (`sourcedir`).
* For example, if `entry.path()` represents the file `C:/source/dir/file.txt`, and `sourcedir` is `C:/source`, then `fs::relative(entry.path(), sourcedir)` will return `dir/file.txt`, which is the relative path of the file with respect to the `C:/source` directory.
* This relative path is useful when copying files to another directory (`destinationdir`) because it allows you to maintain the directory structure of the source directory within the destination directory.