

Lecture 8: Files Advertising

Bart Iver van Blokland

PSA: Halfway evaluation

- We don't use a reference group in this course, and instead rely on your feedback to improve the course
- Should only take a few minutes to give some ratings, but text responses tend to be most useful for making improvements

Thank you so much in advance for your time! <3



PSA: Project description will be published tomorrow

- Counts as the final three assignments of the course
- Build a program from the ground up, with some requirements for what the code must include
- The best projects will receive a prize!
- HIGHLY recommended even though most will have enough assignments approved by that time. Really helps you to get experience with using C++ for your own projects in the future.

Do you remember?

- What is a namespace and what is it used for?
- What is the difference between a struct and a class?
- What is a constructor?
- What is an enum class and what is it used for?

```
struct Slide {
    presentation::SlideType type = SlideType::SLIDE;
    bool autoAdvance = false;

    bool showSlides = true;
    SlidePositioning slideImagePositioning;
    std::string musicToPlay = "";

    ore::RelativeTransformation cameraPosition;
    ore::Transition cameraTransition;

    std::vector<ore::gl::Light> lightPositions;
    std::vector<ore::gl::Light> shadowLightPositions;
    std::vector<presentation::MeshOnSlide> meshes;
    std::vector<presentation::GraphicsObject> graphics;
};
```

```

struct Slide {
    presentation::SlideType type = SlideType::SLIDE;
    bool autoAdvance = false;

    bool showSlides = true;
    SlidePositioning slideImagePositioning;
    std::string musicToPlay = "";

```

```

enum class SlideType {
    SLIDE, DEMONSTRATION
};

```

```

    ore::RelativeTransformation cameraPosition;
    ore::Transition cameraTransition;

```

```

    std::vector<ore::gl::Light> lightPositions;

```

```

    std::vector<ore::Transition> transitions;
    std::vector<ore::gl::Light> lights;
    std::vector<ore::Transition> transitions;
};

namespace ore {
    struct Transition {
        TransitionType type = TransitionType::INSTANT;
        float durationSeconds = 0;
    };
}

```

```
class GameEntryLoadScreen {  
private:  
    ore::scene::ShaderNode shaderNode;  
    ore::gl::GeometryBuffer planeBuffer;  
    ore::scene::OrthographicCamera camera;  
    ore::resources::Material barMaterial;  
    ore::scene::MaterialNode barMaterialNode;  
    ore::scene::GeometryNode node;  
public:  
    void init(ore::resources::ResourceCache *cache);  
    void draw(float progress);  
    void destroy();  
};
```

Today



Files



Today

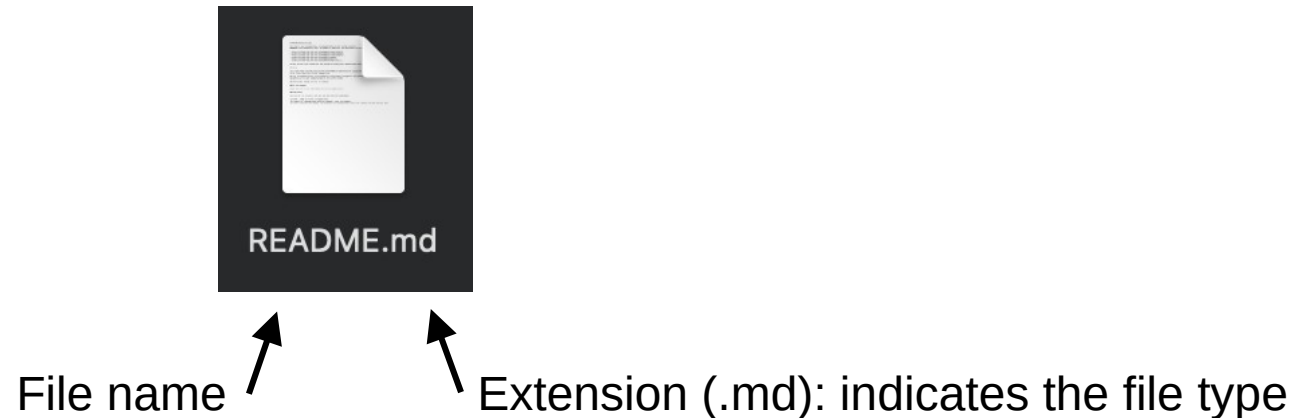
- **Overall objective:**
 - **How do we read a formatted file?**
- **Motivation:**
 - **In practical use, input data of a program is basically always read in from a file rather than the terminal**
- **In VS Code:**
 - Create new project > Lectures > Lecture 08 > Task – mystery images**

Today

- **Understanding files**

Files

- Purpose:
 - Storing data for future use
 - Mechanism to exchange data between programs
- Can be arbitrarily small or large (empty to many gigabytes in size)
- Can only be created in a directory that already exists



File formats

A file is nothing more than a sequence of data

The contents of a file usually follow a specific format, identified by the file's extension, which programs that support the format can interpret

Web page (.html)

```
1 <!DOCTYPE html>
2 <html class="ltr yui3-js-enabled gecko js firefox firefox109 firefox109-0 mac secure" dir="ltr"
  lang="nb-NO"><head>
3   <title>NTNU: Norges teknisk-naturvitenskapelige universitet - NTNU</title>
4   <meta content="initial-scale=1.0, width=device-width" name="viewport">
5
6   <meta name="mobile-web-app-capable" content="yes">
7   <meta name="application-name" content="NTNU">
8
9   <meta name="apple-mobile-web-app-capable" content="yes">
10  <meta name="apple-mobile-web-app-title" content="NTNU">
11  <meta name="apple-mobile-web-app-status-bar-style" content="default">
12
13  <link rel="apple-touch-icon" href="https://www.ntnu.no/ntnu-theme/images/
```

Document (.pdf)

```
1 %PDF-1.6
2 %âãÏÓ
3 1 0 obj
4 <</Metadata 2 0 R/OCProperties<</D<</ON[7 0 R]/Order 8 0 R/RBGroups[]>>/OCGs[7 0 R]>>/Pages 3 0
  R/Type/Catalog>>
5 endobj
6 2 0 obj
7 <</Length 62334/Subtype/XML/Type/Metadata>>stream
8 <?xml:packet begin="ï»¿" id="W5M0MpCehiHzreSzNTczkc9d"?>
9 <x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="Adobe XMP Core 7.2-c000 79.1b65a79, 2022/06/
  13-17:46:14">
10   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
11     <rdf:Description rdf:about=""
12       xmlns:dc="http://purl.org/dc/elements/1.1/"
13       xmlns:xmp="http://ns.adobe.com/xap/1.0/"
```

File paths

- A file path describes where a file or folder is located on your computer
- Can be absolute or relative
 - Absolute path:
 - A path to a file or directory that starts at the root of the file system
 - Relative path:
 - A path to a file or directory that starts elsewhere (usually the directory from which your program is run)

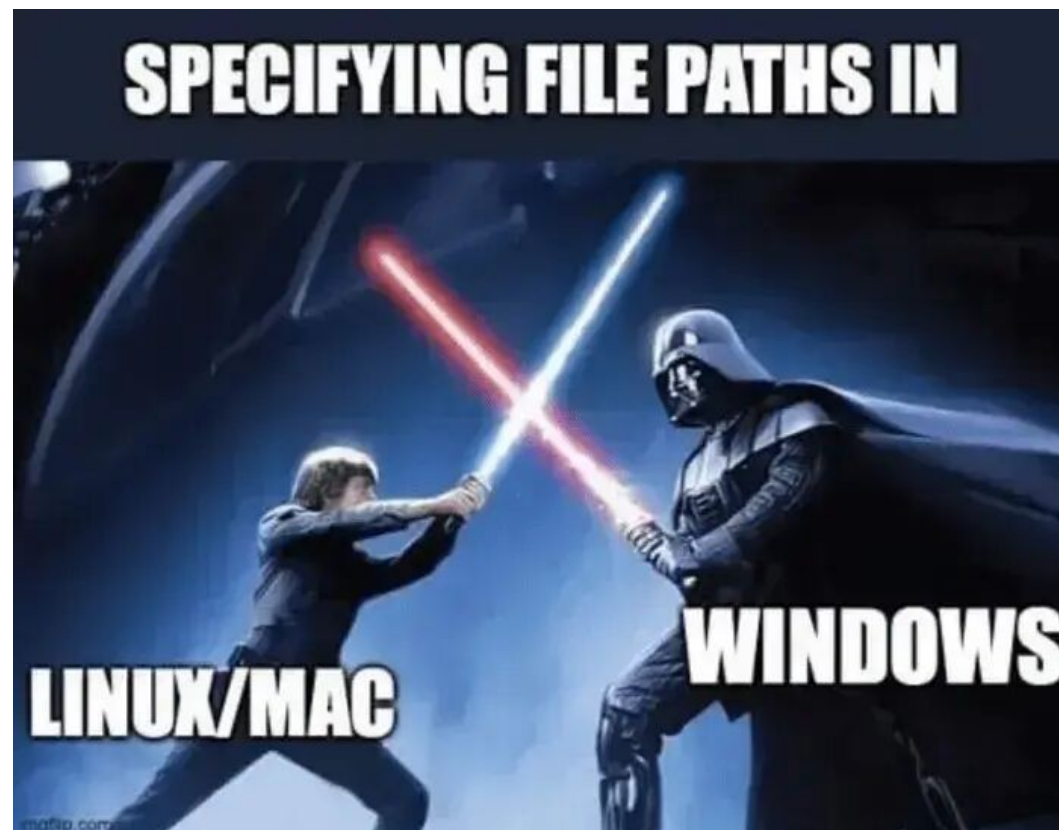
Macintosh HD > Users > bart > git > TDT4102-resources > templates > _folder_Lectures > _folder_Lecture 08 > Task - myster

Examples: paths

- Absolute path:
 - Windows: `C:/Users/user/Documents/workspaces/dev`
 - MacOS: `/Users/user/Documents/workspaces/dev`
- Relative path:
(does not start with a drive letter on Windows (e.g. C:) or a / on linux or macOS, and thus starts in the directory the program is run from)
 - Windows: `../../workspaces/dev`
 - MacOS: `../../workspaces/dev`



The .. means «go one directory up»



Windows used to only allow \ when specifying file paths.

Nowadays / is also supported, and you **really** should use it:

- Your code works across all operating systems
- Need to type \\ in strings, which is a hassle

Today

- Understanding files
- **std::filesystem**

std::filesystem

NOTE: std::filesystem
is not in the book!

- Part of the standard library intended to work with files
- Requires that you `#include` the `<filesystem>` header

std::filesystem::path

- Stores a path to a file or directory
 - Usually folder names separated by /
- `std::string` can also be used, but `std::filesystem::path` communicates that the variable is intended to store a path

Declaring a variable:

```
std::filesystem::path pathToFile{"input/temps.txt"};
```

Declaring a path step by step:

```
std::filesystem::path inputDirectory {"input"};  
std::filesystem::path inputFile =  
    inputDirectory / "measurements" / "temps.txt";
```

Absolute and Relative paths

- Convert relative to absolute path:

```
std::filesystem::path relative {"builddir"};  
std::filesystem::path absPath = std::filesystem::absolute(relative);
```

- Get the program's working directory (where it is run from):

```
std::filesystem::path workDir = std::filesystem::current_path();
```

- Check if two paths are equivalent:

```
std::filesystem::path relative {"builddir"};  
std::filesystem::path absPath = std::filesystem::absolute(relative);
```

```
bool areEquivalent = std::filesystem::equivalent(relative, absPath);  
std::cout << areEquivalent << std::endl; // prints '1'
```

Path utilities

- Check if file or directory exists:

```
std::filesystem::path filePath {"builddir/build.ninja"};  
if(std::filesystem::exists(filePath)) {  
    std::cout << "File exists!" << std::endl;  
}
```

- Print a path:

```
std::filesystem::path buildDirectory {"builddir"};  
std::cout << buildDirectory.string() << std::endl;
```

std::filesystem utilities

- Copy a file or directory:

```
std::filesystem::path sourceFile {"main.cpp"};  
std::filesystem::path destinationFile {"other.cpp"};  
std::filesystem::copy(sourceFile, destinationFile,  
                      std::filesystem::copy_options::recursive);
```



Not required, but recommended for directories

- Create a directory (including nested ones):

```
std::filesystem::path directoryToCreate {"src/core/main"};  
std::filesystem::create_directories(directoryToCreate);
```

std::filesystem utilities

- Delete a file or directory:

```
std::filesystem::path directoryToDelete {"builddir"};  
std::filesystem::remove_all(directoryToDelete);
```

- Rename a file or directory:

```
std::filesystem::path currentPath {"main.cpp"};  
std::filesystem::path destinationPath {"bettermain.cpp"};  
std::filesystem::rename(currentPath, destinationPath);
```

- Get the size of a file in bytes:

```
std::filesystem::path pathToFile {"main.cpp"};  
unsigned long fileSize = std::filesystem::file_size(pathToFile);  
std::cout << "File size: " << fileSize << std::endl;
```


Today

- Understanding files
- `std::filesystem`
- **Reading files**

Reminder: std::cin

- Opposite of cout, reads from the terminal
 - Reads one «word» at a time that is separated with whitespace
 - If you entered multiple words, they are «queued up», and read the next time you call std::cin

```
std::string message;
```

```
std::cin >> message;
```

A terminal window with a black background and white text. The text 'Well hello there' is displayed, followed by a vertical cursor bar, indicating that the input has been read by std::cin.

```
Well hello there
```

Reminder: std::cin

- Opposite of cout, reads from the terminal
 - Reads one «word» at a time that is separated with whitespace
 - If you entered multiple words, they are «queued up», and read the next time you call std::cin

```
std::string message;
```

```
std::cin >> message;
```

Contents of std::cin queue:

«Well hello there!»

Reminder: std::cin

- Opposite of cout, reads from the terminal
 - Reads one «word» at a time that is separated with whitespace
 - If you entered multiple words, they are «queued up», and read the next time you call std::cin

```
std::string message;
```

Contents of std::cin queue:

```
std::cin >> message; ← «Well» — «hello there!»
```

Reminder: std::cin

- Opposite of cout, reads from the terminal
 - Reads one «word» at a time that is separated with whitespace
 - If you entered multiple words, they are «queued up», and read the next time you call std::cin

```
std::string message;
```

Contents of std::cin queue:

```
std::cin >> message;  
std::cout << message << std::endl;  
    // Prints: Well
```

«hello there!»

Reminder: std::cin

- Opposite of cout, reads from the terminal
 - Reads one «word» at a time that is separated with whitespace
 - If you entered multiple words, they are «queued up», and read the next time you call std::cin

```
std::string message;
```

Contents of std::cin queue:

```
std::cin >> message;  
std::cout << message << std::endl;  
// Prints: Well
```

«hello there!»

```
std::cin >> message;  
std::cout << message << std::endl;  
// Prints: hello
```

«hello»

«there!»

Demonstration: `std::ifstream`

Input file stream (std::ifstream)

- Used for reading files
- Requires the `<fstream>` header to be included
- Used in the same way as `std::cin`, but you queue up the entire file

```
std::filesystem::path inputFilePath {"measurements.txt"};  
std::ifstream inputStream {inputFilePath};
```

```
double temperature;  
double humidity;  
double downfall;  
inputStream >> temperature;  
// Left in the queue: 0.6 4.7  
inputStream >> humidity >> downfall  
// File has been read  
std::cout << downfall << std::endl; //prints 4.7
```



measurements.txt

1	25.7
2	0.6 4.7

Good practice: check for errors

- `std::istream` will not automatically tell you if it failed to open a file
 - Can happen if the file does not exist
 - Use the `fail()` function to determine if there's a problem

```
std::filesystem::path inputFilePath {"measurements.txt"};
std::ifstream inputStream {inputFilePath};

if(inputStream.fail()) {
    std::cout << "Oh dear!" << std::endl;
    return 1;
}
```

Task: Read the image1.txt file

- What to do:
 - Open a `std::ifstream` to the file
 - Open a `TDT4102::AnimationWindow`
 - Read the file, and draw each command to the screen
 - Remember to use `wait_for_close()` at the end
- Format:
 - First line is an integer with the number of commands
 - The remaining lines are drawing commands. We only have a command to draw a line between two points for now:
line [x1] [y1] [x2] [y2]
- Done? Try image2.txt. It also has the 'color' command that, if turned on, changes the line command to: line [x1] [y1] [x2] [y2] [colour]
 - `TDT4102::Color` can be constructed using the colour value

Today

- Understanding files
- `std::filesystem`
- Reading files
- **Writing files**

Output file stream (std::ofstream)

- Used for writing files.
 - If the file does not yet exist, it is created automatically
 - The containing directory must already exist
- Also requires the `<fstream>` header to be included
- Writing to an output file works the same as `std::cout`!

```
std::filesystem::path filePath {"results.txt"};  
std::ofstream outputStream {filePath};
```

```
outputStream << "This will be in the output file" << std::endl;
```

Using std::ofstream with std::println

- Tip: you can also use println() to write to an output file stream by passing the stream as an additional parameter!

```
std::ofstream out{"hello.txt"};  
int a = 10;  
std::println(out, "The value stored in a is: {}", a);
```



≡ hello.txt

1 The value stored in a is: 10

2

Today

- Understanding files
- `std::filesystem`
- Reading files
- Writing files
- **String stream**

Stringstream

- Stringstreams are both input and output streams
 - You can use output operators (<<) to add text into the queue
 - You can use input operators (>>) to read 'words' from the queue

```
std::stringstream stream;  
stream << "This text is put in the stream queue" << std::endl;
```

```
std::string word;  
stream >> word;  
std::cout << word << std::endl; // Prints: «word»  
// Stream now contains «Text is put in the stream queue»
```

Stringstream – Use cases

- First major use case: reading a file line by line
 - Or alternatively, you have an existing string from somewhere that contains relevant information that you need to interpret (parse).

```
std::ifstream inputStream{"input.txt"};
std::string line;
while(std::getline(inputStream, line)) {
    std::cout << "Read line in file: " << line << std::endl;
    std::stringstream lineStream;
    lineStream << line;

    // Interpret line here
    int a;
    int b;
    lineStream >> a;
    lineStream >> b;
}
```


Stringstream – str()

- The str() method returns a string of everything that has been written into the stream
 - Important: does not *remove* any data from the stream!
 - Independent of the queue; still includes words that have been read

```
std::stringstream stream;
stream << "This text is put in the stream queue" << std::endl;

std::string word;
stream >> word;
std::cout << word << std::endl; // Prints: «word»
// Stream queue now contains «text is put in the stream queue»

std::cout << stream.str() << std::endl;
// Prints: This text is put in the stream queue
```

stringstream – Use cases

- Second major use case: fast way of building long strings
 - Appending lots of strings is slow
 - Also can make it easier when you have to build strings from many different data types. You can use << instead of having to call to_string()

```
std::string text = std::to_string(a) + " " + std::to_string(b) + " "  
                  + std::to_string(c) + " " + std::to_string(d);
```

```
std::stringstream stream;  
stream << a << " " << b << " " << c << " " << d;  
// Or:  
std::println(stream, "{} {} {} {}", a, b, c, d);
```

Task: what is printed by these?

```
std::stringstream stream;
int a = 5;
stream << 2;
stream << a;
stream << 1;
stream >> a;
std::cout << a << std::endl;
```

```
std::stringstream stream;
stream << "abc" << std::endl;
stream << "def" << std::endl;
std::string text;
stream >> text;
std::cout << text << std::endl;
```

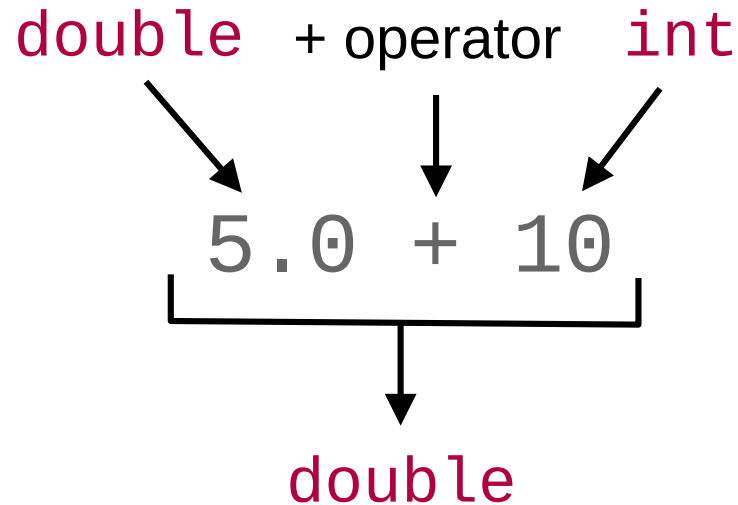
```
std::stringstream stream;
for(int i = 0; i < 5; i++) {
    if(i == 2) {
        std::cout << stream.str() << std::endl;
    }
    stream << i;
}
std::cout << stream.str() << std::endl;
```

Today

- Understanding files
- `std::filesystem`
- Reading files
- Writing files
- String stream
- **Operator overloading**

Operator overloading

You can think of an operator as a function that takes two parameters and returns another value



Operator overloading: syntax

value1 > value2

```
dataType operator>(dataType operand1, dataType operand2) {  
    // Implement the operator here  
}
```

The data type of the result of the operator
(for > that is usually **bool**)

Each dataType can be a different type!

Operators are functions

- Operators are normal functions, except:
 - The name must be **operator** with the desired operator appended to it (e.g. **operator*** or **operator!=**)
 - They are called by using the operator

```
std::string operator+(std::string text, int number) {  
    return text + std::to_string(number);  
}
```

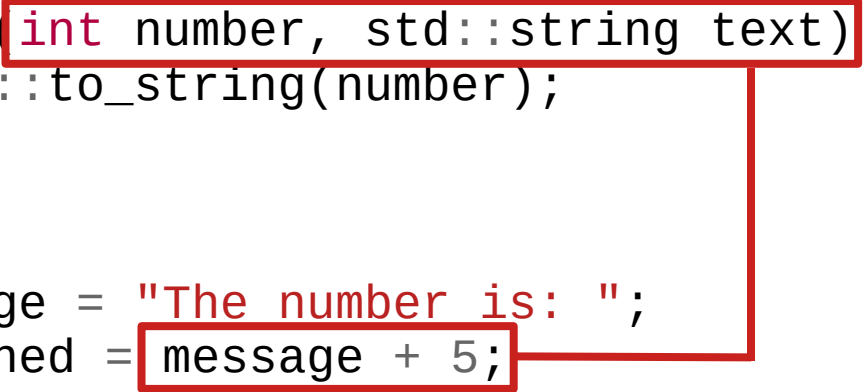
```
int main() {  
    std::string message = "The number is: ";  
    std::string combined = message + 5;  
    return 0;  
}
```

Using the operator calls the operator function!

Operators: Order matters

- The parameter order matters!
 - The example below will not compile, as the operator is only defined for «int + string», not «string + int»

```
std::string operator+(int number, std::string text) {  
    return text + std::to_string(number);  
}  
  
int main() {  
    std::string message = "The number is: ";  
    std::string combined = message + 5;  
    return 0;  
}
```



Operators: There are many!

```
type operator+ (type a, type b) {}  
type operator- (type a, type b) {}  
type operator* (type a, type b) {}  
type operator/ (type a, type b) {}  
type operator% (type a, type b) {}  
type operator^ (type a, type b) {}  
type operator& (type a, type b) {}  
type operator| (type a, type b) {}  
type operator, (type a, type b) {}  
type operator>> (type a, type b) {}  
type operator<< (type a, type b) {}  
type operator~ (type a) {}  
type operator! (type a) {}  
type operator++ (type a) {}  
type operator-- (type a) {}
```

```
type operator== (type a, type b) {}  
type operator!= (type a, type b) {}  
type operator&& (type a, type b) {}  
type operator|| (type a, type b) {}  
type operator< (type a, type b) {}  
type operator> (type a, type b) {}  
type operator<= (type a, type b) {}  
type operator>= (type a, type b) {}
```

For all of these, type can be replaced with **any** data type!

(exception: operators with data types that already exist, like `int + int`)

Using any data type with streams

Using operator overloading, we can use any data type with `std::cout` or `std::ofstream`!

```
struct Quote {  
    std::string text = "If your code is blurry you may not C#";  
    std::string whoSaidIt = "I";  
};
```

Data type that you want
to use with `std::cout` ↓

```
std::ostream& operator<< (std::ostream& stream, Quote quote) {  
    stream << "Quote: " << quote.whoSaidIt << " said \"  
        << quote.text << "\"";  
    return stream;  
}
```

Use the stream
variable as you would
use `std::cout`

Remember to return the stream
(very common mistake on exam answers)

Using any data type with streams

For using `std::cin` and `std::ifstream`, the **operator>>** function can be used.

```
struct Quote {  
    std::string text = "If your code is blurry you may not C#";  
    std::string whoSaidIt = "I";  
};
```

```
std::istream& operator>> (std::istream& stream, Quote& quote) {  
    stream >> quote.whoSaidIt;  
    std::getline(stream, quote.text);  
    return stream;  
}
```

Use `std::getline()` if you want to read a line of text

Remember to return the stream object!

It is important the value being read is passed by reference!

Task

- Let's clean up our program a bit!
 - Define a >> operator that can read a TDT4102::Point from std::cin or a file
 - Define a << operator that can print a TDT4102::Point to std::cout or file
 - Use both operators somewhere in your program

```
std::istream& operator>> (std::istream& stream, Quote& quote) {  
    stream >> quote.whoSaidIt;  
    return stream;  
}  
  
std::ostream& operator<< (std::ostream& stream, Quote quote) {  
    stream << "Quote: " << quote.text;  
    return stream;  
}
```

Spoiler alert!



Using any data type with streams

Printing out private fields of a class is also possible, although it requires declaring the **operator**<< function as a **friend** (explained in a later lecture)

```
class Point {  
    double x = 0;  
    double y = 0;  
    friend std::ostream& operator<< (std::ostream& stream,  
                                     Point point);  
};  
  
std::ostream& operator<< (std::ostream& stream, Point point) {  
    stream << "[" << point.x << ", " << point.y << "];"  
    return stream;  
}
```

Operators: Can be member functions

- When declaring an operand as a member function, the containing class becomes the operator's first parameter

```
struct Point {  
    double x = 0;  
    double y = 0;
```

```
    Point operator+ (Point b) {  
        return {x + b.x, y + b.y};  
    }
```

```
};
```

```
Point operator+(Point a, Point b) {  
    Point sum {a.x + b.x, a.y + b.y};  
    return sum;  
}
```

These are equivalent!



Operators: There are even more!

```
struct vec3 {  
    vec3 operator[] (vec3 index) {}  
    vec3 operator() (vec3 other) {}  
    vec3 operator= (vec3 other) {}  
    vec3 operator+= (vec3 other) {}  
    vec3 operator-= (vec3 other) {}  
    vec3 operator*= (vec3 other) {}  
    vec3 operator/= (vec3 other) {}  
    vec3 operator%= (vec3 other) {}  
    vec3 operator^= (vec3 other) {}  
    vec3 operator&= (vec3 other) {}  
    vec3 operator|= (vec3 other) {}  
    vec3 operator>>= (vec3 other) {}  
    vec3 operator<<= (vec3 other) {}  
};
```

.. But these can **only** be declared as a member function!

As before, vec3 can be replaced with any other data type!

Operators: Tidbits

- Operator overloading extends the language beyond what it is normally able to do.
- For some operators (such as `>>` and `<<` for reading and writing to streams) you usually want to take in the parameters as (const) references.
- As an operator is just another function, it can return any data type, including **void** where that makes sense (for example `+=`).

Today

- Understanding files
- `std::filesystem`
- Reading files
- Operator overloading

Next week

- Memory addresses
- Pointers
- Manual memory management