Lecture 8:
Files Advertising

Bart Iver van Blokland

(Rune Sætre)

Insperaøving 1

- Insperaøving 1:
  - Week 9 (27.02 – 03.03)
  - No lectures and no assignment that week
  - Not mandatory, but highly recommended

- **Insperaøving 2: Mandatory to be able to take the exam**

Details are in the announcement posted on Blackboard
(also sent out by mail)

Follow along on Blackboard for any additional information

Last time

- **Object-Oriented Programing**
- Classes
- Enumerations (enum)

What's an object?

We have already seen a number of them previously!

```
std::vector
std::array
std::string
std::random_device
std::default_random_engine
std::uniform_int_distribution

TDT4102::AnimationWindow
```

More concretely: objects are a combination of data (variables) hidden inside the object, and functions that modify the data in specific ways.

Procedure vs. Object-Oriented Programming

The two styles differ in where program data is stored, and how that program data can be modified.

- Procedure-Oriented Programming:
  - Data is stored inside functions
  - Functions apply modifications on data provided to them as a parameter

- Object-Oriented Programming:
  - Data is hidden within objects
  - Objects have functions that modify the data within

Example: Procedure-Oriented Programming

- Procedure-Oriented Programming:
  - Data is stored inside functions
  - Functions apply modifications on data provided to them as a parameter

```
void increment(int& number) {
    number++;
}

int main() {
    int number = 0;
    increment(number);
    return 0;
}
```

Example: Object-Oriented Programming

- Object-Oriented Programming:
  - Data is hidden within objects
  - Objects have functions that modify the data within

```
int main() {
    AnimationWindow window;
    window.draw_circle({100, 100}, 50);
    window.wait_for_close();
    return 0;
}
```

Comparison

Object-Oriented Programming

```
int main() {
    AnimationWindow window;
    window.draw_circle({100, 100}, 50);
    window.wait_for_close();
    return 0;
}
```

The **object** controls what happens to the data, and can define if and how it is allowed to be modified
(example: cannot change width)

Procedure-Oriented Programming

```
int main() {
    AnimationWindow window = open_wimdow();
    draw_circle(window, {100, 100}, 50);
    window.width = 50;
    wait_for_close(window);
    return 0;
}
```

The **function** controls what happens to the data, and is free to modify it however it wants

Both can be valid choices

Object-Oriented Programming	**Procedure-Oriented Programming**
Control: The object defines what can and cannot be done with the data that is hidden inside	Freedom: A function can do whatever it needs to with any data it processes
Grouping: Data and the functions modifying that data are located together	Granularity: Each function can be independently reused across the entire program
Modularity: As all data and functionality is abstracted away, it can easily be swapped where necessary.	Flexibility: Very few problems can be abstracted neatly, and functions can ensure each edge case is handled

Last time

- Object-Oriented Programing
- **Classes**
- **Enumerations (enum)**

https://leantoys.com/eng_pl_Shark-Attack-Reaction-Family-Game-Push-The-Tooth-760_5.jpg

Today

- **Files and paths**
- **std::filesystem**
- Streams
- std::unordered_map

https://cdn.shopify.com/s/files/1/1862/6381/products/44Farms_09072021_Filet_Raw_2000x2000px_c7e3bd33-597d-4d6d-a636-18f43ebee3af.jpg?v=1633570320

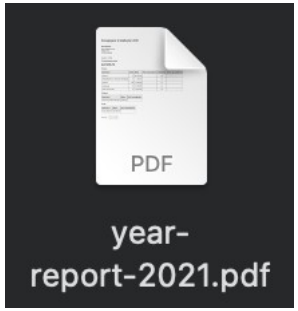
https://img.cuisineaz.com/610x610/2017-03-23/i123634-filets-de-sole-roules-aux-crabes.jpeg

Files

https://www.biltema.no/verktoy/handverktoy/filer/

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 name

Extension (.pdf): indicates the file type

File formats

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)

```
<!DOCTYPE html>
<html class="ltr yui3-js-enabled gecko js firefox firefox109 firefox109-0 mac secure" dir="ltr" lang="nb-NO"><head>
    <title>NTNU: Norges teknisk-naturvitenskapelige universitet - NTNU</title>
    <meta content="initial-scale=1.0, width=device-width" name="viewport">

    <meta name="mobile-web-app-capable" content="yes">
    <meta name="application-name" content="NTNU">

    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="apple-mobile-web-app-title" content="NTNU">
    <meta name="apple-mobile-web-app-status-bar-style" content="default">

    <link rel="apple-touch-icon" href="https://www.ntnu.no/o/ntnu-theme/images/
```

Document (.pdf)

```
%PDF-1.6
%âãÏÓ
1 0 obj
<</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>>
endobj
2 0 obj
<</Length 62334/Subtype/XML/Type/Metadata>>stream
<?xpacket begin="ï»¿" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="Adobe XMP Core 7.2-c000 79.1b65a79, 2022/06/13-17:46:14        ">
   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      <rdf:Description rdf:about=""
            xmlns:dc="http://purl.org/dc/elements/1.1/"
            xmlns:xmp="http://ns.adobe.com/xap/1.0/"
```

std::filesystem::path

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

- Stores a path to a file or directory
- `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

- Absolute path:
  - A path to a file or directory that starts at the root of the file system
    - Windows: `C:/Users/user/Documents/workspaces/dev`
    - MacOS: `/Users/user/Documents/workspaces/dev`

- Relative path:
  - A path to a file or directory that starts in the working directory of the program
    - Windows: `../../workspaces/dev`
    - MacOS: `../../workspaces/dev`

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:

```
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;
```

Today

- Files and paths
- **std::filesystem**
- Streams
- std::unordered_map

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

- Files and paths
- std::filesystem
- **Streams**
- std::unordered_map

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

- 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;
```

Input file stream (std::ifstream)

- Used for reading files
- Also requires the `<fstream>` header to be included
- Used in the same way as `std::cin`!

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

std::string word;
inputStream >> word;
std::cout << word << std::endl;

std::string line;
std::getline(inputStream, line);
std::cout << line << std::endl;
```

Remember: `std::cin` reads one word at a time

The std::getline() function can be used to read a line of text

Spoiler Alert!

https://www.autoguide.com/auto-news/wp-content/uploads/2011/03/Rear-spoiler-1024x682.jpg

Using any data type with streams

Using operator overloading (more details next week), we can use any data type with `std::cout` or `std::ofstream`!

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

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

Put the data type that you want to use with `std::cout` here

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

And remember to return the stream

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 next week)

```
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;
}
```

Using any data type with streams

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

```
struct Point {
    double x = 0;
    double y = 0;
    friend std::istream& operator>> (std::istream& stream,
                                     Point& point)
};

std::istream& operator>> (std::istream& stream, Point& point) {
    stream >> point.x;
    stream >> point.y;
    return stream;
}
```

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

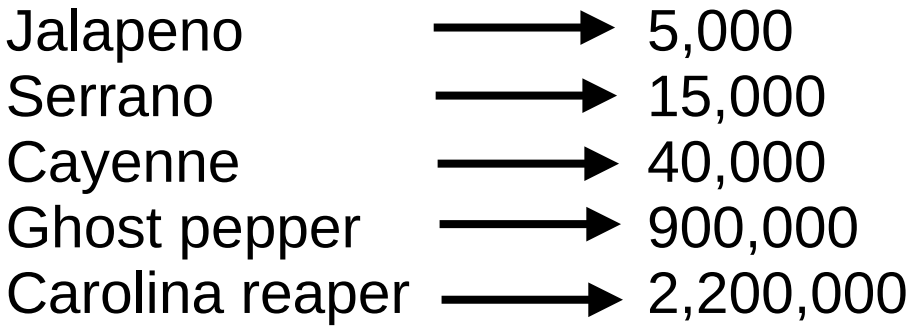

...and now for something completely different

Today

- Files and paths
- std::filesystem
- Streams
- **std::unordered_map**

std::unordered_map and std::map

- Maps connect a unique “key” value to an associated and not necessarily unique “value”
- Include the `<map>` or `<unordered_map>` header
- The C++ equivalent of a dictionary in Python



std::unordered_map and std::map

- Declaring a map:

```
std::unordered_map<std::string, std::string> opposites;
```

Data type of key values

Data type of mapped values

- Insert a value into the map:

```
opposites.insert({"true", "false"});
opposites.insert({"false", "true"});
```

- Read a value:

```
std::string value = opposites.at("false");
std::cout << value << std::endl; // prints true
```

std::unordered_map and std::map

- **Do NOT use the [ ] operator!**

```
std::unordered_map<std::string, double> strengths;
if(strengths["Cayenne"] < 1000) {
    std::cout << "Cayenne is under 1000" << std::endl;
}
```

When using the [ ] operator, if the value being requested is not in the map, *it is created implicitly* even when you are not explicitly assigning a value.

Always use insert() and at() instead.

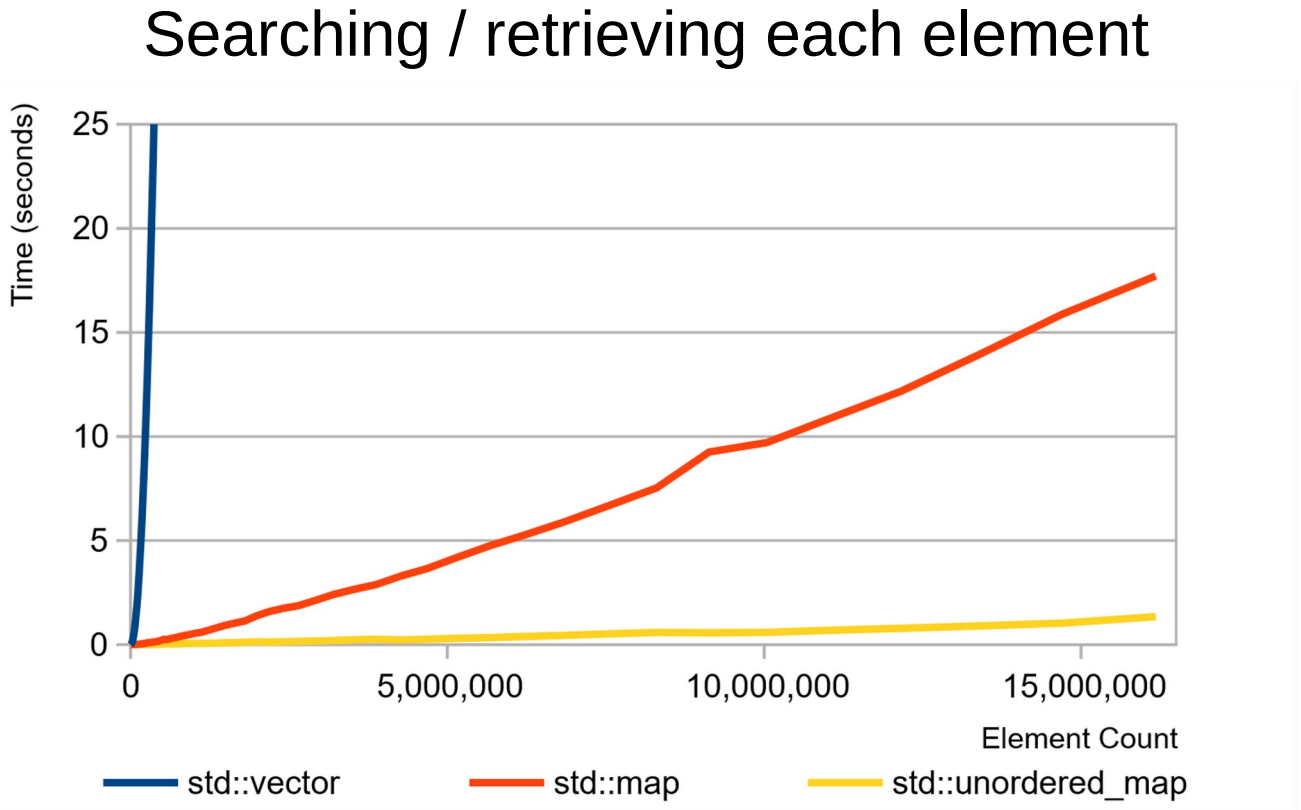
std::unordered_map and std::map

Initialisation and iterating over contents:

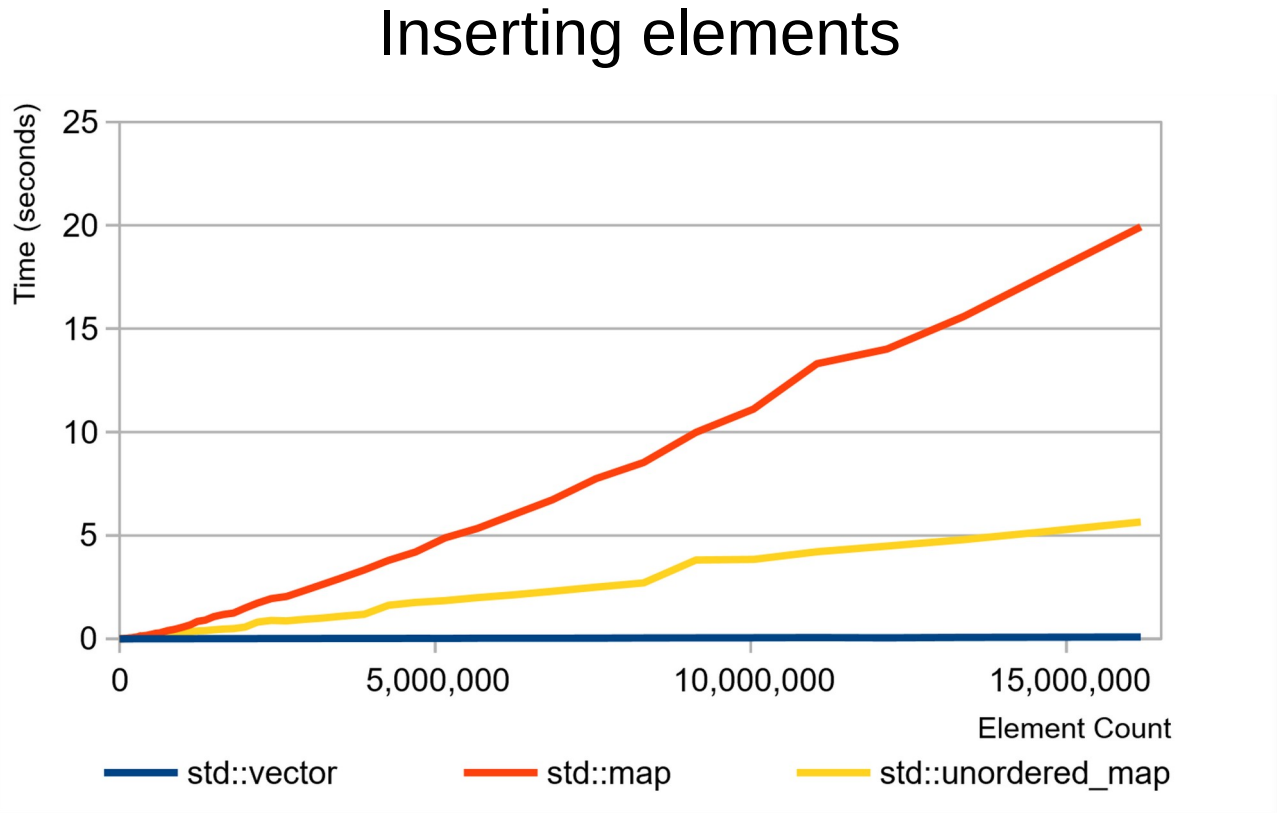
```
std::unordered_map<std::string, double> strengths {
    {"Jalapeno", 5000.0},
    {"Serrano", 15000.0},
    {"Cayenne", 40000.0}
};

for(const std::pair<std::string, double> entry : strengths)
{
    std::cout << entry.first << " -> "
              << entry.second << std::endl;
}
```

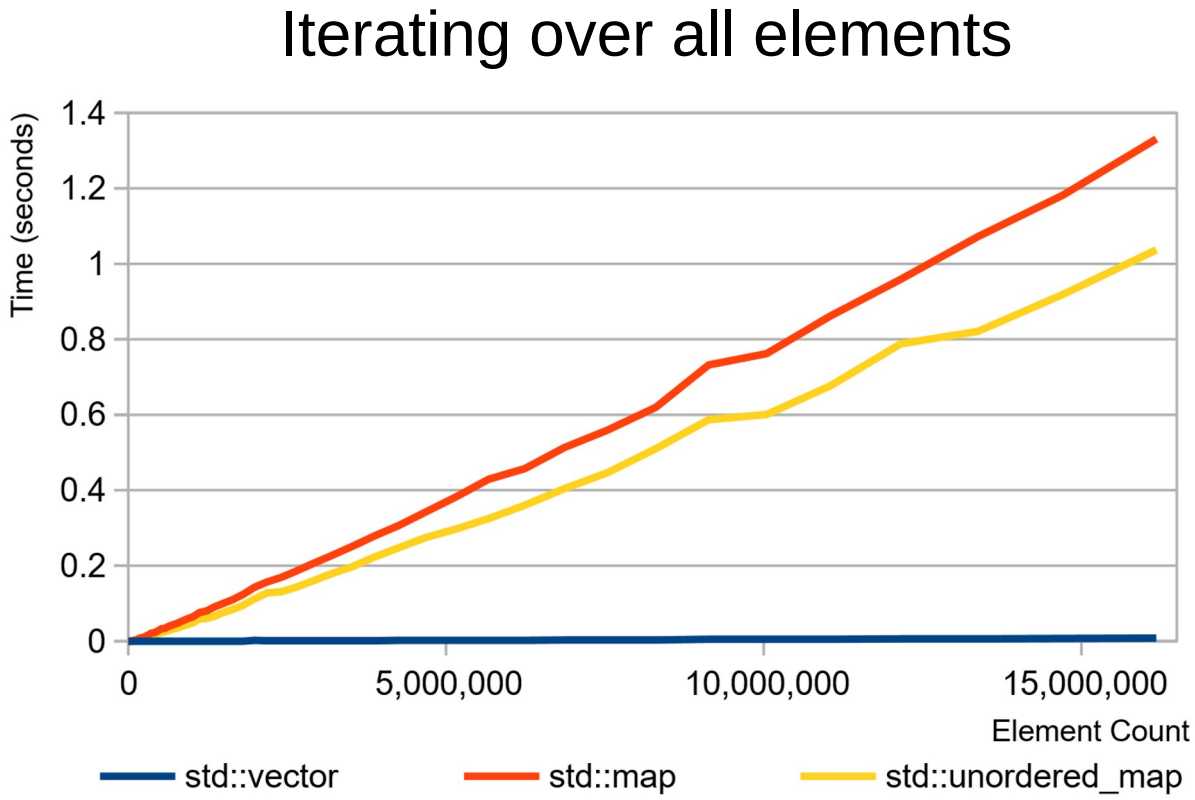
std::unordered_map is usually faster than std::map



std::unordered_map is usually faster than std::map



std::unordered_map is usually faster than std::map



Today

- Files and paths
- std::filesystem
- Streams
- std::unordered_map

Next week

- Inheritance
- Operator overloading
- Friend keyword