

Seminar 02

Separate Compilation, Files, Streams

1. Separate compilation.

- Why do we need it?
 - Separation of the program's logic, readability and future-proofing.
- How does it work?
 - Compiling each .cpp file and linking the object (.o) files (*usually automated*).
- Header files (.h/.hpp)
 - SHOULD contain ONLY **declarations** of functions , structs, *classes*, etc.
- Source files (.cpp)
 - Contain **definitions** of the declared functions, structs, *classes*, etc.
- Include guards
 - Prevents multiple inclusion of header files (*multiple definitions error*)
 - Standard include guard:

```
#ifndef __HEADER_INCLUDED__  
#define __HEADER_INCLUDED__  
    // ..header's code  
#endif
```
 - Non-standard, but widely accepted:

```
#pragma once  
// ..header's code..
```

2. File streams.

- Relative vs absolute path
 - Relative path – path starting from the current directory
 - Absolute path – path starting from the “root” directory (*or a drive*)
 - Note: Windows's paths use ' \ ' as a separator.
To write that in C++ we use ' \\ '.
- Standard IO redirection in the terminal (*bonus*)
- File streams, just like the IO streams - `cout` and `cin`, are streams for input and output of information, but to a certain **file**.
- `std::ifstream`, `std::ofstream` and `std::fstream` are *classes* and we'll be creating objects from these classes to interact with files.
Note: **`cin`** is an object of type *istream*, and **`cout`**, **`cerr`** and **`clog`** - *ostream*

- To use the file stream classes we'll need to include the `fstream` library
`#include <fstream>`
- Text files
 - Used for storing text (*readable data, usually with a '.txt' extension*)
 - Text editors can view the information in the file
 - Easy IO with `>>`, `<<` and `getline` (*just like `cin` and `cout`*)
- Binary files
 - Used for storing objects (*usually with a '.dat' or other extension*)
 - Text editors won't display the file correctly
 - Special methods are used to read/write from/to these files
- Steps when working with files
 1. Ask for a file to be opened.
 2. Check if the file has been opened successfully.
 3. Work with the file.
 4. Close the file as soon as we are done with it.
- File stream creation and methods
 - `std::[i/o]fstream <identifier>;` - creates a file stream object.
 - `open(<file_path>, [flags])` - open file for reading or writing.
 - `close()` - close the file.
 - `eof()` - returns true when the end of the file is reached.
 - `bad()` - returns true if a reading or writing operation fails.
 - `is_open()` - returns true if the file is successfully opened.
 - `write(<place>, <size>)` - write `<size>` number of bytes in a binary file, reading the bytes from `<place>`.
 - `read(<place>, <size>)` - read `<size>` number of bytes from a binary file.

`<place>` is an *address* of type `char*` (or `const char*` for write).
`<size>` is the number of bytes to be written/read.

 - `gcount()` - returns the number of bytes read thus far.
 - `ignore(<num>)` - skips `<num>` bytes from the file.
 - `peek()` - checks the next available character.
 - `tellg()` - returns the position of the **get pointer**.
 - `tellp()` - returns the position of the **put pointer**.
 - `seekg(<pos>, [way])` - changes the position of the **get pointer**.
 - `seekp(<pos>, [way])` - changes the position of the **put pointer**.

`[way]` can be any of `ios_base::beg`, `ios_base::cur`, `ios_base::end`.

Example: `seekp(5, ios_base::end)` means, move the put pointer 5 bytes from the end of the file towards the beginning.

Notes:

When opening files for **writing**, by default *the content of the file is erased*.

When opening a **non-existing** file for **writing**, that file *will be created*.

- Flags when opening files

- Additional options can be added when opening files that are defined in the second parameter of the open method (or in the constructor).

- Flags:

`std::ios::app` - when writing, **append** at the end of the file.

`std::ios::trunc` - when a file has been opened for writing, erase (truncate) its content first before writing.

`std::ios::binary` - open a file in binary mode.

`std::ios::in` - open file for reading (*when an fstream object is used*)

`std::ios::out` - open file for writing (*when an fstream object is used*)

- Note: Multiple flags can be added using the binary OR operator (`|`)

Example:

```
std::fstream file("myFile.bin", std::ios::in | std::ios::binary);
```

- Simple output to a text file example:

```
std::ofstream out("file.txt"); // Step 1
if (!out)
{ // Step 2
    std::cout << "Couldn't open file for writing!" << std::endl;
}
else
{
    out << "This text will go in the file" << std::endl; // Step 3
    out.close(); // Step 4
}
```

- Simple **OUT**put to a binary file example:

```
int num = 42;
char str[MAX_LEN];
strcpy(str, "Test string");

// Step 1
std::ofstream outBin("data.bin", std::ios::binary);

// Step 2
if (!outBin)
{
    std::cout << "Couldn't open file for writing!" << std::endl;
}
```

```

else
{
    // Step 3
    // Writing primitive data types
    outBin.write(reinterpret_cast<const char*>(&num), sizeof(num));

    // Writing C strings involves writing their length first
    // and then writing the actual string
    int len = strlen(str);
    outBin.write(reinterpret_cast<const char*>(&len), sizeof(len));
    outBin.write(str, len);

    // Step 4
    outBin.close();
}

```

- Simple **IN**put to a binary file example:

(Very similar to output, but we've switched to read methods and char* cast instead)

```

int num;
char str[MAX_LEN] = {};

// Step 1
std::ifstream inBin("data.bin", std::ios::binary);

// Step 2
if (!inBin)
{
    std::cout << "Couldn't open file for reading!" << std::endl;
}
else
{
    // Step 3
    // Reading primitive data types
    inBin.read(reinterpret_cast<char*>(&num), sizeof(num));

    // Reading C strings involves reading their length first
    // and then reading the actual string
    int len;
    inBin.read(reinterpret_cast<char*>(&len), sizeof(len));

    if (len < MAX_LEN - 1)
    {
        inBin.read(str, len);
    }
}

```

```
        str[len+1] = '\0';  
    }  
  
    // Step 4  
    inBin.close();  
}
```