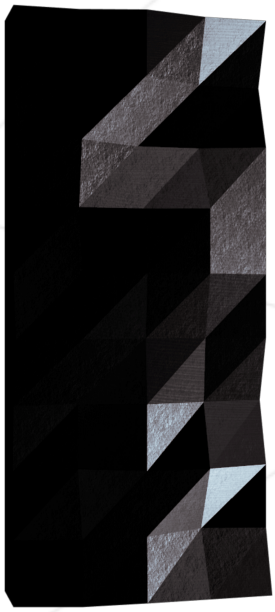

# libftpp

## Further Exploration into Advanced C++

*Summary: This subject aims to introduce you to advanced C++ concepts through the development of complex tools and systems.*

*Version: 1.1*

# Contents

# Chapter I

# Objectives

In this cutting-edge subject, you will embark on an extraordinary journey into the depths of C++ to craft a toolbox that will be your companion for the rest of your C++ projects!

Gone are the days of relying solely on pre-built libraries and frameworks. It's time to take control, build tools tailored to your unique challenges, and uphold the highest standards of code quality.

As you delve into design patterns like Singleton and Observer, you'll begin to see the architecture of effective software unfold before you.

You'll explore thread-safe data structures, elevating your multi-threading capabilities to new heights.

Imagine writing network code so robust that you could practically stream the entirety of the internet through it.

Have you ever been bogged down by complex mathematical algorithms or feared that you might not be able to handle them? Fret not, as you'll create your own customizable vector classes and random number generators, fine-tuned to your every requirement, proving to yourself that you can do it!

This toolbox won't just be a set of classes and methods; it will be a testament to your skill, your creativity, and your mastery of one of the most powerful programming languages to date.

So gear up for this incredible adventure, because by the end, you'll have not just a toolbox but an arsenal of skills that will empower you to tackle any software project in the future.

# Chapter II

# General rules

### Compiling

- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code must compile with at least the flag `-std=c++11`

`Yes, you must use CPP11 or newer.`

### Formatting and naming conventions

- Name your files, classes, functions, member functions and attributes as required in the guidelines.
- Write class names in **SnakeCase** format. Files containing class code will always be named according to the class name. For instance:
  `class_name.hpp`/`class_name.h`, `class_name.cpp`, or `class_name.tpp`. Then, if you have a header file containing the definition of a class "BrickWall" standing for a brick wall, its name will be `brick_wall.hpp`.
- Unless specified otherwise, every output messages must be ended by a new-line character and displayed to the standard output.
- *Goodbye Norminette!* No coding style is enforced in the C++ modules. You can follow your preferred style. However, remember that if your code is difficult for your peer evaluators to understand, they won't be able to grade it. Do your best to write clean and readable code.

### Allowed/Forbidden

You are not coding in C anymore. Time to C++! Therefore:

- You are allowed to use almost everything from the standard library, from at least C++11 up to the newest version of C++. Thus, instead of sticking to what you already know, it would be wise to use the C++ versions of the C functions you are accustomed to, as much as possible.

- However, you cannot use any external libraries beyond the standard library. This means libraries such as `Boost` are forbidden. The following functions are also prohibited: `*printf()`, `*alloc()`, and `free()`. If you use any of these, your grade will be an automatic 0, no exceptions.

### A few design requirements

- Memory leakage occurs in C++ as well. When you allocate memory using the `new` keyword, you must ensure you avoid **memory leaks**.
- Any function implementation placed in a header file (except for function templates) will result in a grade of 0 for that exercise.
- Each of your headers should be usable independently of the others. Therefore, they must include all necessary dependencies. However, you must prevent the issue of double inclusion by adding **include guards**. Failure to do so will result in a grade of 0.

### Read me

- You may add additional files if needed (e.g., to organize your code). Since these assignments are not automatically verified by a program, you are free to do so as long as you submit the mandatory files.

You will need to implement many classes. This may seem tedious unless you can script your favorite text editor.

You have a certain amount of freedom in completing the exercises. However, follow the mandatory rules and don't be lazy-you would miss out on a lot of valuable information! Don't hesitate to read up on theoretical concepts.

# Chapter III

# Mandatory part

## III.1 General structure of your library

In this subject, your ultimate goal is to build a comprehensive C++ toolbox that will serve you well in your future projects. To ensure that your toolbox is both portable and easy to integrate, you are required to structure your project as follows:

- **Makefile:** Your project should include a Makefile that, when executed, produces a static library named `libftpp.a`. This library should bundle all the functionalities you implement throughout this subject. Your Makefile must compile `libftpp.a` using the `-Wall -Wextra -Werror` flags.
- **Header File:** You must provide a unified header file named `libftpp.hpp`, which includes all the necessary headers from your toolbox. This will enable other developers (or your future self) to easily integrate your library into new projects by including just this single header file.
- **Organization:** Apart from these essential components, you are free to organize your codebase as you see fit. Creativity and good structure are encouraged but not strictly enforced. You're free to organize sources and headers however you prefer—the choice is yours.

## III.2 Data Structures

In this section, we will delve deep into the cornerstone of every software: Data Structures. Understanding and implementing efficient data structures is a crucial skill in software development.

We expect you to provide a `data_structures.hpp` file, which will include all the necessary headers you create in this section.

You will create a `Pool` and `Pool::Object` class for resource pooling, as well as a `DataBuffer` class that offers a polymorphic container for storing objects in byte format.

| | Pool and Pool::Object |
|---|---|
| Files to be delivered | `pool.hpp` |
| Description | Manages a collection of reusable templated TType objects, provided to the user via a Pool::Object class that will handle the acquirable pointer, and release it to the pool when needed, calling the destructor of the TType object but without deallocating the memory. |
| Methods | Pool :<br>- `void resize(const size_t& numberOfObjectStored)` : Allocated a certain number of TType object inside the Pool.<br>- `template<typename ... TArgs> Pool::Object<TType> acquire(TArgs&& p_args)` : Create a Pool::Object containing a pre-allocated object, by using the constructor with parameters following TArgs definition.<br>Pool::Object :<br>- TType* operator -> (): return the pointer stored inside the Pool::Object. |
| Hints | Every request/release of a pre-allocated object must be executed by Pool::Object, not by the user !<br>If you don't understand template TArgs, you should look at variadic template, it's cool and elegant |

| | DataBuffer |
|---|---|
| Files to be delivered | `data_buffer.hpp, data_buffer.cpp` |
| Description | A polymorphic container for storing objects as bytes |
| Methods | Templated operator overloads for «, » |
| Hints | Use C++ stream operators for serialization and deserialization. |

## III.3 Design Patterns

Now that you have set up your library folders and Makefile, let's dive into the first section of this subject: the creation of a set of classes following the concepts introduced by ... `The Gang of Four!`.

We expect you to provide a `design_patterns.hpp` file, which will include all the necessary headers you create in this section.

As an example, let's create the following structures!

| | Memento |
|---|---|
| Files to be delivered | `memento.hpp, memento.cpp` |
| Description | Stores the current state of an object. Must be inherited by the "saveable" class. |
| Methods | - Snapshot save() : save the current state of the object<br>- void load(const Memento::Snapshot& state) : load the provided state<br><br>To be saved, the inheriting class must implement the following methods, as private :<br>- `void _saveToSnapshot(Memento::Snapshot& snapshot)` : Save the desired data inside the snapshot.<br>-`void _loadFromSnapshot(Memento::Snapshot& snapshot)` : Load data saved inside the snapshot. |
| Hints | Implementation should allow easy restoration. Maybe you should think of a way to easily save/load data into something like a polymorphic container or something ...<br>Maybe your Memento will need to access those privates methods, _saveToSnapshot and _loadFromSnapshot, i wonder if there is a **friend**ly way to do it ... |

| | Observer |
|---|---|
| Files to be delivered | `observer.hpp` |
| Description | A class Observer, who is templated by TEvent, who allow user to subscribe to events and be notified when thoses specifics events are triggered. |
| Methods | Public methods :<br>`-void subscribe(const TEvent& event, const std::function<void()>& lambda)` : Subscribe a lambda to a specific event.<br>`- void notify(const TEvent& event)` : Will execute every event-subscribed lambda. |
| Hints | Use the Observer pattern to update registered objects when a subject changes. |

| | Singleton |
|---|---|
| Files to be delivered | `singleton.hpp` |
| Description | Ensures a templated TType class has only one instance and provides a way to access it |
| Methods | Public methods :<br>`- TType* instance()` : Return the managed instance of the TType classes.<br>`-template<typename ... TArgs> void instanciate(TArgs&& p_args)` : Setup the managed instance of the class. If the instance is already set, this method must throw an exception. |
| Hints | This class must be set as friend in the inherited class |

| | StateMachine |
|---|---|
| Files to be delivered | `state_machine.hpp` |
| Description | A class StateMachine, templated TState, managing transition between states. |
| Methods | Public methods :<br>- `void addState(const TState& state)` : Add a specific possible state to the object.<br>- `void addTransition(const TState& startState, const TState& finalState, const std::function<void()>& lambda)` : Indicate what to execute uppon specific transition.<br>-`void addAction(const TState& state, const std::function<void()>& lambda)` : Indicate what to execute when the StateMachine is in a specific state.<br>- `void transitionTo(const TState& state)` : Execute the transition to a specific state.<br>- `void update()` : Execute the action registred for the current state. |
| Hints | Handle state transitions cleanly and efficiently. If a transition or an update isn't setup by the user, the StateMachine must throw an exception. |

## III.4 IOStream

Input/Output operations often become bottlenecks in high-performance applications. In this section, you will implement a thread-safe I/O stream with prefixed lines. This will allow you to debug and monitor applications with multiple threads more easily.

| | ThreadSafeIOStream |
|---|---|
| Files to be delivered | `thread_safe_iostream.hpp, thread_safe_iostream.hpp` |
| Description | Thread-safe version of iostream |
| Methods | Public methods :<br>- Operator overloads for «, »<br>- `void setPrefix(const std::string& prefix)` : set the prefix printed in front of the line when using your iostream overload. |
| Hints | Make sure it is thread-safe and thread local and can add a prefix to each line. Don't forget to create an equivalent to the std::cout, to not be forced to create manually your iostream !<br>We expect you to provide something like "thread_local ThreadSafeIOStream threadSafeCout;" inside your header. |

## III.5 Thread

Threading is an essential concept for any developer aiming to write high-performance, scalable applications.

You will build several classes to handle thread-safe queues, promises, and worker pools.

We expect you to provide a `threading.hpp`, which will include all the headers you create in this section.

These tools will be invaluable in your future projects involving concurrent programming.

| | ThreadSafeQueue |
|---|---|
| Files to be delivered | `thread_safe_queue.hpp` |
| Description | A templated TType thread-safe version of a queue |
| Methods | Public methods :<br>- `void push_back(const TType& newElement)` : add an element at the end of the queue.<br>- `void push_front(const TType& newElement)` : add an element in front of the queue.<br>- `TType pop_back(const & newElement)` : extract the last element of the queue.<br>- `TType pop_front(const TType& newElement)` : extract the first element of the queue. |
| Hints | Use mutexes or other mechanisms for thread safety. If users try to pop something from an empty queue, it must throw an exception. |

| | Thread |
|---|---|
| Files to be delivered | `thread.hpp thread.cpp` |
| Description | A wrapper for the std::thread, with a name |
| Methods | Public methods :<br>`-Thread(const std::string& name, std::function<void()> functToExecute)` : A constructor setting up the thread data, waiting for a start call to launch the function.<br>- `void start()` : Launch the thread, executing the function passed as parameter to its constructor.<br>- `void stop()` : stop the thread, joining it correctly. |
| Hints | The thread name should be used by ThreadSafeIOStream and must provide it with a prefix indicating which thread is printing. |

| | WorkerPool |
|---|---|
| Files to be delivered | `worker_pool.hpp, worker_pool.cpp` |
| Description | Manages worker threads to execute jobs. It contain a subclass IJob, describing a basic interface for a job to be executed by the worker pool. |
| Methods | `-void addJob(const std::function<void()>& jobToExecute)` : Insert a new job to be executed by the pool |
| Hints | Threads should be perpetual. |

| | PersistentWorker |
|---|---|
| Files to be delivered | `persistent_worker.hpp persistent_worker.cpp` |
| Description | A thread that continuously performs a set of tasks defined by users. |
| Methods | `-void addTask(const std::string& name, const std::function<void()>& jobToExecute)` : insert a task in the task pool of the worker, naming it<br>`- void removeTask(const std::string& name)` : Remove a job from the pool worker. |
| Hints | Should have a list of tasks to perform in a loop. |

## III.6 Network

Networking is the backbone of the modern world, and learning how to manipulate it will give you a strong advantage in any field.

This section will require you to implement several fundamental networking constructs, like Messages, Connections, and Servers.

We expect you to provide a `network.hpp`, which will include all the headers you create in this section.

You will be crafting the basic building blocks for creating networked applications.

| | Message |
|---|---|
| Files to be delivered | `message.hpp, message.cpp` |
| Description | Handles messages between client and server |
| Methods | - Message(int type) : A constructor taking a int as input, to describe the type of message stored inside the object.<br>- Templated operator overloads for «, ».<br>- `int type()` : return the type of the message. |
| Hints | The classes using those Message must be able to detect which type of message they received, using a Type attribute inside your Message to know how to use them. |

| | Client |
|---|---|
| Files to be delivered | `client.hpp, client.cpp` |
| Description | Client-side networking |
| Methods | -`void connect(const std::string& address, const size_t& port)` : connect to the specific server by address and port.<br>- `void disconnect()` : Disconnect from the server.<br>- `void defineAction(const Message::Type& messageType, const std::function<void(const Message& msg)>& action)` : Subscribe an action to a specific message type.<br>- `void send(const Message& message)` : Send a message to the connected server<br>- `void update()` : Treat every received message since last update call, and execute the action subscribed by the user for each message. |
| Hints | Should work seamlessly with Server and Message classes. |

| | Server |
|---|---|
| Files to be delivered | `server.hpp server.cpp` |
| Description | Server-side networking |
| Methods | - `void start(const size_t& p_port)` : Start the server at the desired port.<br>-`void defineAction(const Message::Type& messageType, const std::function<void(long long& clientID, const Message& msg)>& action)` : Subscribe an action to a specific message type.<br>-`void sendTo(const Message& message, long long clientID)` : Send a message to a specific client ID.<br>-`void sendToArray(const Message& message, std::vector<long long> clientIDs)` : Send a message to a specific set of client ID.<br>- `void sendToAll(const Message& message)` : Send a message to every client IDs currently connected to the server.<br>- `void update()` : Treat every received message since last update call, and execute the action subscribed by the user for each message. |
| Hints | Should manage multiple clients and routes. |

`You can pass this subject without doing the next section.`

## III.7 Mathematics

Although C++ already provides a broad range of mathematical functions and libraries, there's value in building your own.

In this section, you'll create classes for 2D and 3D vectors using templated types.

We expect you to provide a `mathematics.hpp`, which will include all the headers you create in this section.

You'll also work on generating pseudo-random numbers and implement 2D Perlin noise to introduce you to some mathematical implementations.

| | IVector2 |
|---|---|
| Files to be delivered | `ivector2.hpp, ivector2.cpp` |
| Description | A 2D vector with templated type |
| Methods | IVector2 must be a struct, holding attributes : - TType x : represent the first coordinate of the vector.<br>- TType y : represent the second coordinate of the vector.<br>Operator:<br>- Operator overloads with other IVector2 for +, -, *, /, ==, !=.<br>Additional:<br>- `float length()` : Return the norm/length of the vector<br>- `IVector2<float> normalize()` : Return the normalized vector from the 2D vector.<br>- `float dot()` : Return the dot product of the 2D vector.<br>- `IVector2 cross()` : return the cross product of the 2D vector. |
| Hints | Use C++ operator overloading for clean and intuitive code. |

| | IVector3 |
|---|---|
| Files to be delivered | `ivector3.hpp, ivector3.cpp` |
| Description | A 3D vector with templated TType |
| Methods | IVector3 must be a struct, holding attributes : - TType x : represent the first coordinate of the vector.<br>- TType y : represent the second coordinate of the vector.<br>- TType z : represent the third coordinate of the vector.<br>Operator:<br>- Operator overloads with other IVector3 for +, -, *, /, ==, !=.<br>Additional:<br>- `float length()` : Return the norm/length of the vector<br>- `IVector3<float> normalize()` : Return the normalized vector from the 3D vector.<br>- `float dot()` : Return the dot product of the 3D vector.<br>- `IVector3 cross()` : return the cross product of the 3D vector. |
| Hints | Extend functionalities using C++ operator overloading. |

The following structures are `pseudo-random generators` (You may want to check what this is ;)).
As such, the combination of Seed and Coordinates must always return the same result every time they are called. Obviously, if you change the seed or change the coordinates, the result must change.

| | Random2DCoordinateGenerator |
|---|---|
| Files to be delivered | `random_2D_coordinate_generator.hpp`, `random_2D_coordinate_generator.cpp` |
| Description | Generates pseudo-random numbers based on 2D coordinates |
| Methods | - `long long seed()` : Return the seed of the generator<br>`-long long operator()(const long long& x, const long long& y)` : Generate a pseudo random number using the two values passed as parameter. |
| Hints | Use 2D coordinates as part of the random number generation. Overload the function call operator (). |

| | PerlinNoise2D |
|---|---|
| Files to be delivered | `perlin_noise_2D.hpp, perlin_noise_2D.cpp` |
| Description | Generates 2D Perlin noise |
| Methods | - `float sample(x, y)` : return a perlin value for the coordinates provided to the function. |
| Hints | Overload the function call operator () for generating noise based on coordinates. |

# Chapter IV

# Bonuses

We strongly encourage you to create any class you think is relevant.

For example, here are some ideas for possible bonuses!

- `Timer`
  Allows you to set a duration and check if it has timed out, using system time.
- `Chronometer`
  Allows you to count durations, using system time.
- `Application and Widget`
  Allows you to create custom applications with interesting behaviors.
- `ObservableValue`
  Notifies any subscribers upon modification.

This list is not exhaustive, so we encourage you to add anything you think is relevant. However, you must justify your choices and provide tests to demonstrate the functionality of these classes you create.

Each class you create will give you 5 bonus points.

`The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been completed in its entirety and works without malfunction. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.`

# Chapter V

# Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your files to ensure they are correct.

You are encouraged to organize your code/includes as you see fit, but you must provide a Makefile that builds a library named `libftpp.a`, and you must provide an include file named `libftpp.hpp` at the root of your repository.