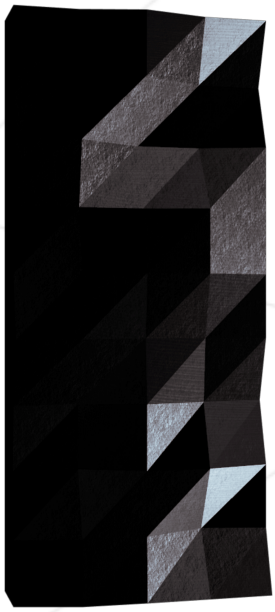

libftpp

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: 1.3

Contents

I Objectives **2**

II General rules **3**

III Mandatory part **5**

III.1 General structure of your library 5
III.2 Data Structures 6
III.3 Design Patterns 8
III.4 IOStream 11
III.5 Thread 12
III.6 Network 14
III.7 Mathematics 16

IV Bonuses **19**

V Submission and peer-evaluation **20**

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. Now it's time to take control, build tools tailored to your unique challenges, and uphold the highest standards of code quality.

As you explore design patterns like Singleton and Observer, you'll start to see how the architecture of effective software unfold before you.

You'll dive into thread-safe data structures, pushing your multi-threading capabilities to new heights.

Imagine writing network code so robust that it could handle streaming the entire internet.

Have complex mathematical algorithms ever seemed daunting? Fear not! You'll create your own customizable vector classes and random number generators, fine-tuned to your specific needs—proving to yourself that you're more than capable.

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

So gear up for an incredible adventure, because by the end, you won't just have a toolbox! you'll have an arsenal of skills that will empower you to take on 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 C++11 or later.`

Formatting and naming conventions

- Name your files, classes, functions, member functions, and attributes as outlined in the guidelines.
- Write class names in **PascalCase** format. Write method names in **camelCase** format. Files containing class code should always be named in **snake_case** according to the class name. For instance:
  `class_name.hpp`/`class_name.h`, `class_name.cpp`, or `class_name.tpp`. So, if you have a header file containing the definition of a class "BrickWall" representing a brick wall, the file name will be `brick_wall.hpp`.
- Unless specified otherwise, every output messages must end with a new-line character and be 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. So, Do your best to write clean and readable code.

Allowed/Forbidden

You are not coding in C anymore. It's 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++. So, 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()`. Using any of these will be result in an automatic 0 for your grade, 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. Thus, 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 project, 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 project. 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 dive deep into the most fundamental aspects of software: Data Structures. Understanding and implementing efficient data structures is a crucial skill in software development.

You are expected to provide a `data_structures.hpp` file, which will include all the necessary headers created in this section.

You will implement a `Pool` class and a `Pool::Object` class for resource pooling, as well as a `DataBuffer` class that serves as 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. This class handles the acquirable pointer, and releases it back to the pool when needed, calling the destructor of the TType object but without deallocating the memory.
Methods	Pool : `-void resize(const size_t& numberOfObjectStored)`: Allocates a certain number of TType objects withing the Pool. `-template<typename ... TArgs> Pool::Object<TType> acquire(TArgs&& p_args)`: Creates a Pool::Object containing a pre-allocated object, using the constructor with parameters as defined by TArgs definition. Pool::Object : `-TType* operator -> ()`: Returns the pointer stored withing the Pool::Object.
Hints	Every requests and releases of a pre-allocated objects must be handled by Pool::Object, not by the user ! If you're unfamilliar with template TArgs, you should look at variadic templates, they're cool and elegant

	DataBuffer
Files to be delivered	`data_buffer.hpp, data_buffer.cpp`
Description	A polymorphic container for storing objects in byte format.
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 project: creating a set of classes based on the concepts introduced by ... `The Gang of Four!`.

You are expected to provide a `design_patterns.hpp` file, which will include all the necessary headers created 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. -`void load(const Memento::Snapshot& state)`: Loads the provided state. To be saved, the inheriting class must implement the following methods, as private : -`void _saveToSnapshot(Memento::Snapshot& snapshot)` const: Saves the desired data into the snapshot. -`void _loadFromSnapshot(Memento::Snapshot& snapshot)`: Loads data from the snapshot.
Hints	The implementation should allow easy restoration. You might want to think of a way to easily save/load data into something like a polymorphic container ... Also, your Memento may need access to these private methods, _saveToSnapshot and _loadFromSnapshot. I wonder if there is a **friend**ly way to do it ...

	Observer
Files to be delivered	`observer.hpp`
Description	An Observer class templated by TEvent, which allows users to subscribe to events and be notified when those specifics events are triggered.
Methods	Public methods : `-void subscribe(const TEvent& event, const std::function<void()>& lambda)`: Subscribes a lambda to a specific event. `-void notify(const TEvent& event)`: Executes all lambdas subscribed to the event.
Hints	Use the Observer pattern to update registered objects when a subject changes.

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

	StateMachine
Files to be delivered	`state_machine.hpp`
Description	A `StateMachine` class templated by `TState`, managing transitions between states.
Methods	Public methods : `-void addState(const TState& state)`: Adds a specific possible state to the object. `-void addTransition(const TState& startState, const TState& finalState, const std::function<void()>& lambda)`: Specifies what to execute uppon a specific transition. `-void addAction(const TState& state, const std::function<void()>& lambda)`: Specifies what to execute when the StateMachine is in a specific state. `-void transitionTo(const TState& state)`: Executes the transition to a specific state. `-void update()`: Executes the action registred for the current state.
Hints	Handle state transitions cleanly and efficiently. If a transition or an update isn't set up 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.cpp, thread_safe_iostream.hpp`
Description	Thread-safe version of `iostream`.
Methods	-
Hints	-

- Methods Public methods :

-Operator overloads for «, »
-`void setPrefix(const std::string& prefix)`: Sets the prefix printed before the line when using your `iostream` overload.
-`template<typename T> void prompt(const std::string& question, T& dest);`

Ensure it is thread-safe and thread-local, adding a prefix to each line. Additionally, provide an equivalent to std::cout so there's no need to create a custom iostream. We expect you to include something like thread_local ThreadSafeIOStream threadSafeCout; in your header file.

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.

You are expected to provide a `threading.hpp` file, 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 : -`void push_back(const TType& newElement)`: Adds an element at the end of the queue. -`void push_front(const TType& newElement)`: Adds an element to the front of the queue. -`TType pop_back(const & newElement)`: Extracts the last element from the queue. -`TType pop_front(const TType& newElement)`: Extracts the first element from the queue.
Hints	Use mutexes or other mechanisms for thread safety. If users try to pop an element 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 : -`Thread(const std::string& name, std::function<void()> functToExecute)`: A constructor that stets up the thread data, waiting for a `start()` call to launch the function. -`void start()`: Launches the thread, executing the function passed as a parameter to its constructor. -`void stop()`: Stops the thread, joining it properly.
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 `IJobs`, which describes interface for a job to be executed by the worker pool.
Methods	`-void addJob(const std::function<void()>& jobToExecute)`: Inserts a new job to be executed by the pool
Hints	Threads should run perpetually.

	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)`: Inserts a task into the worker's task pool, assigning it a name `-void removeTask(const std::string& name)`: Removes a task from the worker's task pool.
Hints	Should maintain a list of tasks to perform in a loop.

III.6 Network

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

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

You are expected to provide a `network.hpp` file, which should include all the headers created in this section.

You will be building the basic components 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 that takes an integer as input to describe the type of message stored inside the object. -Templated operator overloads for «, ». -`int type()`: Returns the type of the message.
Hints	Classes using `'Message'` must be able to detect the type of message they receive, using a `'Type'` attribute inside `'Message'` to determine how to handle it.

	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. -`void disconnect()`: Disconnect from the server. -`void defineAction(const Message::Type& messageType, const std::function<void(const Message& msg)>& action)`: Subscribe an action to a specific message type. -`void send(const Message& message)`: Send a message to the connected server -`void update()`: Process all 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 specified port. `-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. `-void sendTo(const Message& message, long long clientID)`: Send a message to a specific client ID. `-void sendToArray(const Message& message, std::vector<long long> clientIDs)`: Send a message to a specific set of client IDs. `-void sendToAll(const Message& message)`: Send a message to all clients currently connected to the server. `-void update()`: Process all received messages since last update call and execute the action subscribed by the user for each message.
Hints	Should manage multiple clients and mesage routing.

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.

You are expected to provide a `mathematics.hpp` file, which should include all the headers created in this section.

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

	IVector2
Files to be delivered	`ivector2.hpp, ivector2.cpp`
Description	A 2D vector with a templated type
Methods	IVector2 must be a struct, holding the following attributes: -`TType x`: Represents the first coordinate of the vector. -`TType y`: Represents the second coordinate of the vector. Operator: -Operator overloads with other `IVector2` for `+`, `-`, `*`, `/`, `==`, `!=`. Additional: -`float length()`: Returns the norm/length of the vector -`IVector2<float> normalize()`: Returns the normalized version of the 2D vector. -`float dot()`: Returns the dot product of the 2D vector. -`IVector2 cross()`: Returns 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 a templated `TType`
Methods	IVector3 must be a struct, holding the following attributes: - `TType x`: Represents the first coordinate of the vector. - `TType y`: Represents the second coordinate of the vector. - `TType z`: Represents the third coordinate of the vector. Operator: - Operator overloads with other IVector3 for `+`, `-`, `*`, `/`, `==`, `!=`. Additional: -`float length()`: Returns the norm/length of the vector -`IVector3<float> normalize()`: Returns the normalized version of the 3D vector. -`float dot()`: Returns the dot product of the 3D vector. -`IVector3 cross()`: Returns the cross product of the 3D vector.
Hints	Extend functionality 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()`: Returns the seed of the generator. `-long long operator()(const long long& x, const long long& y)`: Generates a pseudo-random number using the two values passed as parameters.
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)`: Returns a perlin noise value for the provides coordinates.
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.

Here are some ideas for possible bonus classes!

- `Timer`
  Allows you to set a duration and check if it has timed out, using system time.
- `Chronometer`
  Allows you to measure durations, using system time.
- `Application and Widget`
  Allows you to create custom applications with interesting behaviors.
- `ObservableValue`
  Notifies subscribers when the value is modified.

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

Each class you create will earn you 5 bonus points.

`The bonus part will only be evaluated if the mandatory part is PERFECT. Perfect means the mandatory part has been completed in its entirety and works flawlessly. If you do not meet ALL the mandatory requirements, your bonus part will not be evaluated.`

Chapter V

Submission and peer-evaluation

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Be sure to double-check the names of your files to ensure they are correct.

You are free to organize your code and 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.