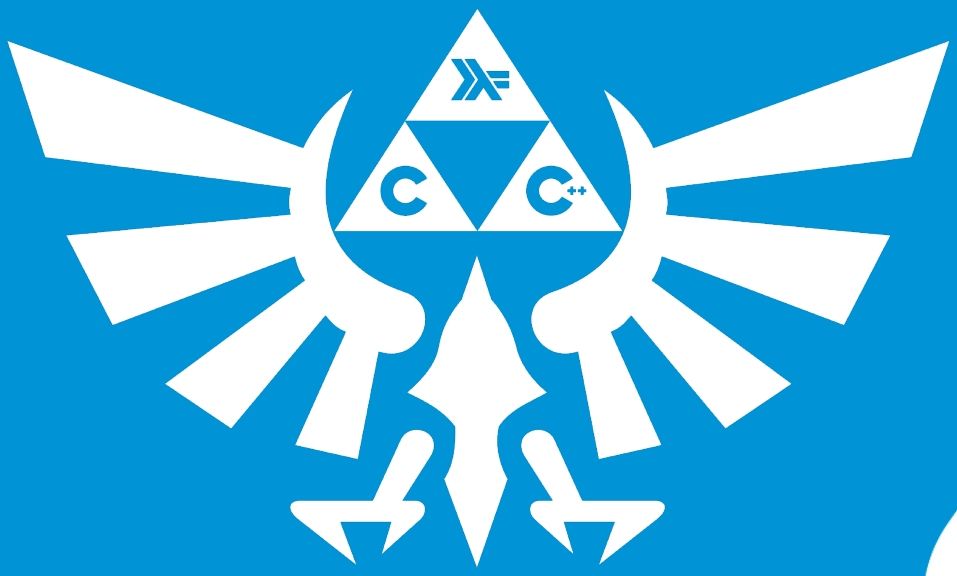
{EPITECH}

DAY 13

TEMPLATES & STL



DAY 13

All your exercises will be compiled with `g++` **and the** `-Wall -Wextra -Werror -std=c++20` **flags**, unless specified otherwise.

All output goes to the standard output, and must be ended by a newline, unless specified otherwise.

None of your files must contain a `main` function, unless specified otherwise.
We will use our own `main` functions to compile and test your code. It will include your header files.

There are no subdirectories to create for each exercice. Every file must be at the root of the repository.

Read the examples CAREFULLY. They might require things that weren't mentioned in the subject…

If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercices because you're lazy, and leave at 2PM, you **WILL** have problems. Do not tempt the devil.

The `*alloc`, `free`, `*printf`, `open` and `fopen` functions, as well as the `using namespace` keyword, are forbidden in C++. By the way, `friend` is forbidden too, as well as any library except the standard one.

Unit Tests

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the **“How to write Unit Tests”** document on the intranet, available here.

Exercise 0 - Algorithm

Turn in : `Algorithm.hpp`

> *This exercisme is not an exercisme about platypuses.*
> *Thank you for your understanding.*

Write the following function templates :

- ✓ `swap` : swaps the value of its two parameters.
- ✓ `min` : returns the smallest of its two parameters.
- ✓ `max` : returns the biggest of its two parameters.
- ✓ `clamp` : takes three parameters (value, min and max), returns the value clamped between min and max.

These templates should generate functions that can be called with any type of parameter, as long as they have the same type and **provide a** < **comparison operator**.

Here is a sample `main` function and its expected output :

```cpp
int main(void)
{
    int a = 42;
    int b = 21;

    ::swap(a, b);
    std::cout << "a = " << a << ", b = " << b << std::endl;
    std::cout << "min(a, b) = " << ::min(a, b) << std::endl;
    std::cout << "max(a, b) = " << ::max(a, b) << std::endl;
    std::cout << "clamp(0, a, b) = " << ::clamp(0, a, b) << std::endl;

    std::string c = "ghi";
    std::string d = "abc";

    ::swap(c, d);
    std::cout << "c = " << c << ", d = " << d << std::endl;
    std::cout << "min(c, d) = " << ::min(c, d) << std::endl;
    std::cout << "max(c, d) = " << ::max(c, d) << std::endl;
    std::cout << "clamp(\"def\", c, d) = " << ::clamp(std::string("def"), c, d) << std::
        endl;

    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp && ./a.out
a = 21, b = 42
min(a, b) = 21
max(a, b) = 42
clamp(0, a, b) = 21
c = abc, d = ghi
min(c, d) = abc
max(c, d) = ghi
clamp(“def”, c, d) = def
```

Exercise 1 - Array

Turn in: `Array.hpp`

> *This exercisme is not an exercisme about exorcism.*
> *Thank you for your understanding.*

Create an `Array` templated class encapsulating arrays.
It must take its Type and Size as template parameters.

Provide overloads of its `[]` operator to access and modify (when non-const) its content.
If the position if out of range, you must throw an exception with the following `what` message :

```
Out of range
```

You must also provide an overloading of the << operator to a `std::ostream&` to write the content of the array to the stream using the following format :

```
[x1, x2, x3, ..., xN]
```

Implement the following methods :

```
std::size_t size() const;
```

Returns the size of the array;

```
void forEach(const std::function<void(const Type&)>& task) const;
```

Call the given `task` function for each element in the `Array`.

```
template<typename U>
Array<U, Size> convert(const std::function<U(const Type&)>& converter) const;
```

Convert the current `Array<Type, Size>` to a `Array<U, Size>` using the `converter` conversion function for each element.

Here is a sample `main` function and its expected output :

```cpp
int main(void)
{
    Array<int, 3>    array;

    try {
        array[0] = 21;
        array[1] = 42;
        array[2] = 84;
        array[3] = 84;
    }
    catch (const std::exception& e) {
        std::cout << e.what() << std::endl;
    }

    std::cout << array << std::endl;
    array
        .convert<float>([](const int& v) { return static_cast<float>(v) / 10.f; })
        .forEach([](const float& v) { std::cout << v << std::endl; });
    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp && ./a.out
Out of range
[21, 42, 84]
2.1
4.2
8.4
```

Exercise 2 - Stack

Turn in: `Stack.cpp/hpp`

> *This exercisme is not an exercisme about cycligm.*
> *Thank you for your understanding.*

Create a `Stack` class encapsulating a `std::stack<double>`.
Implement the following methods :

```
void push(double value);
```

Push a new `value` at the top of the stack.

```
double pop();
```

Remove the top value and return it.

```
double top() const;
```

Return the value at the top of the stack.

```
void add();
void sub();
void mul();
void div();
```

Pop two values from the stack and push the result of the corresponding operation back to the stack.

Errors must be handled through exception.
Create a `Stack::Error` class inheriting from `std::exception` that returns the following `what` :

- ✓ `pop` or `top` on an empty stack : `Empty stack`
- ✓ Missing `add`, `sub`, `mul` or `div` operand : `Not enough operands`

When an error occurs, the `Stack` is left unchanged.

Here is a sample `main` function and its expected output :

```
int main(void)
{
    Stack stack;

    stack.push(42.21);
    stack.push(84.42);
    stack.push(21.12);
    stack.add();
    stack.div();

    try {
        stack.mul();
    }
    catch (const std::exception& e) {
        std::cout << e.what() << std::endl;
    }

    std::cout << stack.top() << std::endl;
    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp Stack.cpp && ./a.out
Not enough operands
2.50036
```

Exercise 3 - UniquePointer

Turn in : `UniquePointer.hpp`

> *This exercisme is not an exercisme about video gasme.*
> *Thank you for your understanding.*

Remember your encapsulation of pointers ? We're gonna make a generic one using template !

Create a `UniquePointer` templated class taking as template parameter the `Type` of the pointer stored.
The `UniquePointer` is in charge of deleting this pointer at destruction.
It is not copyable to avoid having the same pointer stored in multiple `UniquePointer`.

You must handle the following cases:

```
UniquePointer<int> ptr1(new int(42));
UniquePointer<int> ptr2;        // Hold a nullptr
UniquePointer<int> ptr3(ptr1); // DOES NOT COMPILE

ptr2 = ptr1;                    // DOES NOT COMPILE
ptr2 = new int(84);
ptr1 = new int(21);             // Old int* is deleted
```

You must also provide the following methods :

```
void reset();
```

Deletes the stored pointer, replacing it with `nullptr`.

```
Type *get();
```

Returns the stored pointer.

It must also be possible to access the stored pointer using the -> operator.

Obviously, the usage of `std::unique_ptr` is forbidden.

Here is a sample `main` function and its expected output :

```
class Example
{
private:
    int _id;

public:
    Example(int id) : _id(id) { std::cout << "#" << _id << " construction" << std::endl; }
    ~Example() { std::cout << "#" << _id << " destruction" << std::endl; }

    void method() const { std::cout << "#" << _id << " method" << std::endl; }
};

int main(void)
{
    UniquePointer<Example>  ptr1(new Example(1));
    UniquePointer<Example>  ptr2(new Example(2));

    ptr1.reset();
    ptr2 = new Example(3);
    ptr2.get()->method();
    ptr2->method();

    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp && ./a.out
#1 construction
#2 construction
#1 destruction
#3 construction
#2 destruction
#3 method
#3 method
#3 destruction
```

Exercise 4 - Command

Turn in: `Command.cpp/hpp`

> *This exercisme is not an exercisme about spaghetiti.*
> *Thank you for your understanding.*

In this exercise, we will create a class to handle simple commands in the form of `std::string`.
We will associate `std::string`s to `std::function`s call.

Create a `Command` class.
This class contains an `std::map` associating command name to the function to execute.

Implement the following methods :

```
void    registerCommand(const std::string& name, const std::function<void()>& function);
```

Register a new `name` command in the map to the `function`.
If `name` is already registered, throw a `Command::Error` exception with the following `what` message : `Already registered command`

```
void executeCommand(const std::string& name);
```

Execute the function associated to the command `name`.
If `name` is not registered, throw a `Command::Error` exception with the following `what` message : `Unknow command`

`std::map`

Here is a sample `main` function and its expected output :

```
int main(void)
{
    Command command;
    Stack stack;

    try {
        command.registerCommand("push", [&stack]() { stack.push(4.2); });
        command.registerCommand("display", [&stack]() { std::cout << stack.top() << std::
            endl; });
        command.registerCommand("add", [&stack]() { stack.add(); });
        command.registerCommand("sub", [&stack]() { stack.sub(); });
        command.registerCommand("mul", [&stack]() { stack.mul(); });
        command.registerCommand("div", [&stack]() { stack.div(); });
        command.registerCommand("display", []() {});
    }
    catch (const std::exception& e) {
        std::cout << e.what() << std::endl;
    }

    try {
        command.executeCommand("push");
        command.executeCommand("push");
        command.executeCommand("push");
        command.executeCommand("add");
        command.executeCommand("div");
        command.executeCommand("display");
        command.executeCommand("nau");
    }
    catch (const std::exception& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp Command.cpp Stack.cpp &&
./a.out
Already registered command
2
Unknow command
```

Exercise 5 - SharedPointer

Turn in : `SharedPointer.hpp`

> *This exercisme is not an exercisme about homoeopathy.*
> *Thank you for your understanding.*

`UniquePointer` is cool, but wouldn't it be AWESOME if it was copyable ? Your pointer being used everywhere in your code and being automatically deleted when you drop the last reference on it. Let's make your dream a reality !

Create a `SharedPointer` templated class taking as template parameter the `Type` of the pointer stored. A `SharedPointer` is just like a `UniquePointer` except it is copyable.

You must handle the following cases:

```
int *ptr = new int(42);
SharedPointer<int> ptr1(ptr);  // ptr is now owned by ptr1
SharedPointer<int> ptr2;       // ptr2 hold a nullptr
SharedPointer<int> ptr3(ptr1); // ptr is now owned by both ptr1 and ptr3

ptr2 = ptr1;                    // ptr is now owned by both ptr1, ptr2 and ptr3
ptr1 = new int(84);             // ptr is now owned by ptr2 and ptr3, ptr1 hold a new
    pointer
ptr2.reset();                   // ptr is now owned by ptr3 only
ptr3.reset();                   // last reference of ptr is dropped, ptr is deleted
```

You'll need to share a use counter between every copy of your SharedPointer. The implementation is up to you !

Obviously, the usage of `std::shared_ptr` is forbidden.

Here is a sample `main` function and its expected output:

```cpp
class Example
{
private:
    int _id;

public:
    Example(int id) : _id(id) { std::cout << "#" << _id << " construction" << std::endl; }
    ~Example() { std::cout << "#" << _id << " destruction" << std::endl; }

    void method() const { std::cout << "#" << _id << " method" << std::endl; }
};

int main(void)
{
    SharedPointer<Example> ptr1(new Example(1));
    SharedPointer<Example> ptr2(ptr1);
    SharedPointer<Example> ptr3;

    ptr1.reset();
    ptr3 = ptr2;
    ptr2 = new Example(2);
    ptr2.get()->method();
    ptr3->method();

    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp && ./a.out
#1 construction
#2 construction
#2 method
#1 method
#1 destruction
#2 destruction
```

Exercise 6 - Shell

Turn in: `Shell.cpp/hpp`

> *This exercisme is not an exercisme about waterploo.*
> *Thank you for your understanding.*

Create a `Shell` class encapsulating the parsing of command line.
We will use it to extract lines from a `std::istream` and converting each word to the desired `Type` using `std::stringstream`.

The constructor take a reference to a `std::istream` as parameter.

```
Shell(std::istream& stream);
```

This is the `stream` you will extract lines from. At construction, no line should be extracted.

```
void next();
```

Get the next line from the stream ready to be parsed.
Throw a Shell::Error exception with the following `what` message in case of failure :

- ✓ EOF : `End of input`
- ✓ Otherwise : `Input failed`

```
template<typename T>
T extract();
```

Extract and convert the next word of the current line to the requested parameter `Type`.
Throw a `Shell::Warning` exception with the following `what` message in case of failure : `Invalid conversion`

Here is a sample `main` function and its expected output:

```
int main(void)
{
    Shell shell(std::cin);
    Stack stack;
    Command command;

    command.registerCommand("push", [&shell, &stack]() { stack.push(shell.extract<double
        >()); });
    command.registerCommand("display", [&stack]() { std::cout << stack.top() << std::endl;
         });
    command.registerCommand("add", [&stack]() { stack.add(); });
    command.registerCommand("sub", [&stack]() { stack.sub(); });
    command.registerCommand("mul", [&stack]() { stack.mul(); });
    command.registerCommand("div", [&stack]() { stack.div(); });

    while (true) {
        try {
            std::cout << "> " << std::flush;
            shell.next();
            command.executeCommand(shell.extract<std::string>());
        }
        catch (const Shell::Error& e) {
            std::cout << e.what() << std::endl;
            break;
        }
        catch (const std::exception& e) {
            std::cout << e.what() << std::endl;
            continue;
        }
    }
    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp Shell.cpp Command.cpp
Stack.cpp && ./a.out
> push 4.56
> push abc
Invalid conversion
> push 42
> div
> display
9.21053
> mul
Not enough operands
> CTRL+D End of input
```

Exercise 7 - Matrix

Turn in: `Matrix.hpp`

> *This exercisme is not an exercisme about the Petit Lu cookies.*
> *Thank you for your understanding.*

Create a `Matrix` templated class taking two `unsigned int` as template parameter, the first being the number of rows and the second the number of columns of the matrix.
A `Matrix` is composed of `double` zero-ed at construction.

A `Matrix` is basically an `Array` of `Array`.

We can access the values of the matrix using the parenthesis operator :

```
double operator()(unsigned int row, unsigned int col) const;
double& operator()(unsigned int row, unsigned int col);
```

We must be able to multiply matrices using `*` and `*=` operators.

You worked with matrices in the 102architect, remember ?

Implement an operator overloading of `<<` to write `Matrix` to a `std::ostream`.
Here is an example of the output formating for a `Matrix<3, 4>` :

```
[[x0, x1, x2, x3], [y0, y1, y2, y3], [z0, z1, z2, z3]]
```

Definitely look like an `Array` of `Array`...

Here is a sample `main` function and its expected output:

```cpp
int main(void)
{
    Matrix<3, 1> point;
    Matrix<3, 3> translation;
    Matrix<3, 3> rotation;

    point(0, 0) = 3;
    point(1, 0) = 2;
    point(2, 0) = 1;
    std::cout << "P: " << point << std::endl;

    translation(0, 0) = translation(1, 1) = translation(2, 2) = 1.0;
    translation(0, 2) = 4;
    translation(1, 2) = 2;
    point = translation * point;
    std::cout << "T: " << translation << std::endl;
    std::cout << "P: " << point << std::endl;

    rotation(0, 0) = +std::cos(std::numbers::pi / 2);
    rotation(0, 1) = -std::sin(std::numbers::pi / 2);
    rotation(1, 0) = +std::sin(std::numbers::pi / 2);
    rotation(1, 1) = +std::cos(std::numbers::pi / 2);
    rotation(2, 2) = 1;
    point = rotation * point;
    std::cout << "R: " << rotation << std::endl;
    std::cout << "P: " << point << std::endl;

    return 0;
}
```

Terminal

```
~/B-PDG-300> g++ -Wall -Wextra -Werror -std=c++20 main.cpp && ./a.out
P: [[3], [2], [1]]
T: [[1, 0, 4],[0, 1, 2], [0, 0, 1]]
P: [[7], [4], [1]]
R: [[6.12323e-17, -1, 0], [1, 6.12323e-17, 0], [0, 0, 1]]
P: [[-4], [7], [1]]
```

v 3