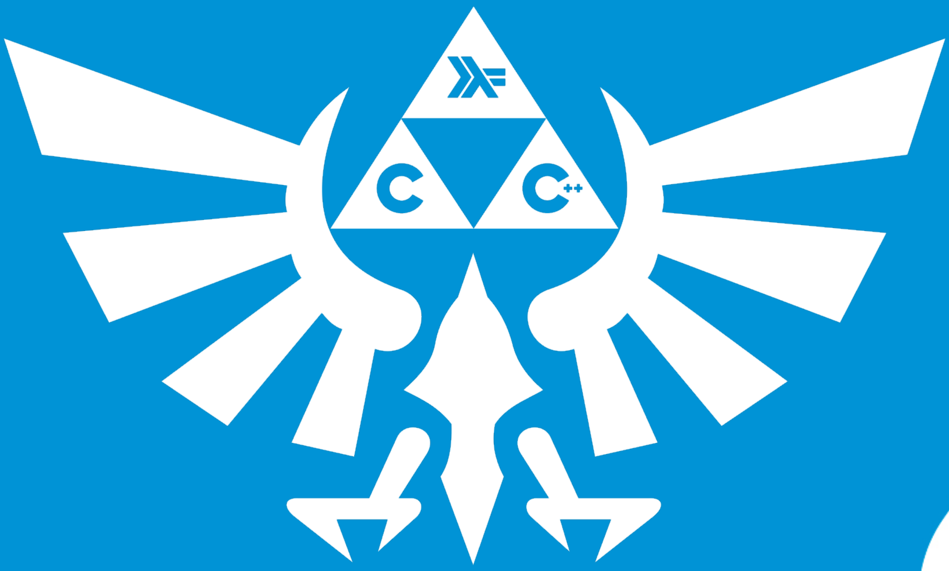
DAY 11

ENCAPSULATE, ENCAPSULATE!!!



DAY 11

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

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.

For them to be executed and evaluated, put a `Makefile` at the root of your directory with the `tests_run` rule as mentionned in the documentation linked above.

Exercise 0 - Listing directories

Turn in : `IDirectoryLister.hpp`, `DirectoryLister.hpp/cpp`

C is a tedious language... Everything has to be done by hand, every return value must be checked for error, pointers are unsafe, and you have to release resources (memory allocations, file descriptors, etc.) yourself! The user being an idiot, we will create strong tools that cannot be broken. We must ensure everything is being taken care of. Every unsafe aspect of the C language must be encapsulated into a class.

Look at the `std::string` *class, isn't it gorgeous ? It's safe, doesn't leak... Perfection.*

Today, you are going to create your own encapsulation of the `opendir` / `readdir` / `closedir` C functions to be able to list files in a directory in C++!

A good encapsulation inherits from an interface to allow for multiple implementations. Create the `IDirectoryLister` with the following methods :

- ✓ `bool open(const std::string& path, bool hidden);`
- ✓ `std::string get();`

Then, implement the interface through the `DirectoryLister` class. The following behavior is expected :

- ✓ `open`: opens the new directory given as parameter. Returns `true` on success. In case of failure, the directory stream becomes invalid, a description of the error (`perror`) is printed on the standard error output and the method returns `false`.
- ✓ `get`: returns the name of the next entry in the current directory, or an empty string if the end of the directory stream is reached or the stream is invalid. Hidden files are ignored when the directory is opened with the `hidden` parameter set to `false`.

Your encapsulation is expected to not leak any `DIR *`.

Provide the two following constructors :

- ✓ `DirectoryLister()`: creates the class without opening a directory.
- ✓ `DirectoryLister(const std::string& path, bool hidden)`: opens the directory with the given parameters.

For safety, it must not be possible to call a copy or move constructor, or an assignement operator.

We know `std::filesystem` exists since `C++17`... However, you **MUST** use `opendir`, `readdir` and `closedir` for this exercise.

The following code should produce the expected output :

```cpp
int main(void)
{
    DirectoryLister dl("./test/", true);

    for (std::string file = dl.get(); !file.empty(); file = dl.get())
        std::cout << file << std::endl;
    dl.open("invalid path", true);
    if (dl.open("./test/", false) == true)
        for (std::string file = dl.get(); !file.empty(); file = dl.get())
            std::cout << file << std::endl;

    return 0;
}
```

Terminal

```
~/B-PDG-300> ls -a ./test/
.
..
.hidden
file1
file2
subdirectory
~/B-PDG-300> ./a.out 2>&1 | cat -e
.$
..$
file1$
.hidden$
file2$
subdirectory$
invalid path: No such file or directory$
file1$
file2$
subdirectory$
```

The order of the filenames extracted is not important.

Exercise 1 - Error management

Turn in: `IDirectoryLister.hpp/cpp`, `DirectoryLister.hpp/cpp`, `SafeDirectoryLister.hpp/cpp`

Returning errors is tedious : we have to check the return value at every step of our code. Empty string, null pointers, ... If the user is not careful, they could crash any time. And we can't even check the return value of a constructor as it desn't have any! We're doomed... ***Except*** *if...*

`SafeDirectoryLister` has the same behavior, except that error are handled through exceptions.

Define the following exception classes inheriting from `std::exception` and implement them in `IDirectoryLister.cpp`:

- ✓ `IDirectoryLister::OpenFailureException`: exception class whose `what` method returns the error message as returned by `strerror`.
- ✓ `IDirectoryLister::NoMoreFileException`: exception class whose `what` method returns `"End of stream"`.

Re-implement `DirectoryLister` using exceptions to handle errors in a new class named `SafeDirectoryLister` . The `open` method will always returns `true` and no longer display an error message. `get` will throw an exception when the end of the directory stream is reached.

The following code must compile and produce the following output :

```cpp
void myLs(const std::string& directory)
{
    try {
        SafeDirectoryLister dl(directory, false);

        std::cout << directory << ":" << std::endl;
        for (std::string file = dl.get(); true; file = dl.get())
            std::cout << file << std::endl;
    } catch (const IDirectoryLister::NoMoreFileException& e) {
        return;
    }
    throw std::runtime_error("should not happen");
}

int main(void)
{
    try {
        myLs("./test/");
        myLs("./not_exist/");
        myLs("./test/");
    } catch (const IDirectoryLister::OpenFailureException& e) {
        std::cerr << "failure: " << e.what() << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "unexpected error: " << e.what() << std::endl;
    }
    return 0;
}
```

Terminal

```
~/B-PDG-300> ./a.out 2>&1 | cat -e
./test/:
file1$
file2$
subdirectory$
failure: No such file or directory$
```

Exercise 2 - Smart pointers, unique pointers

Turn in : UniquePointer.hpp/cpp
Provided files : IObject.hpp

Congratulations, now you've learnt to handle errors in C++! Now, it's time to handle memory allocations. Objects are automatically constructed when declared and destroyed at the closing of their scope. This isn't the case for objects manually allocated using `new`. You're going to hack C++ to leave memory management to the compiler !

In this exercise, all the objects handled by the `UniquePointer` class inherit from the provided `IObject` interface.

The class `UniquePointer` will manage the dynamically allocated object given to it. Create a `UniquePointer` class, and deduce its behavior using the following code example :

```
int main(void)
{
    UniquePointer ptr1;
    UniquePointer ptr2(new TestObject("Eccleston"));
    //UniquePointer ptr3(ptr2); <- Does not compile!

    ptr1 = new TestObject("Tennant");
    ptr2 = new TestObject("Smith");
    ptr1->touch();
    (*ptr2).touch();
    {
        UniquePointer ptr4(new TestObject("Whittaker"));
    }
    ptr1.reset(new TestObject("Capaldi"));
    ptr1.swap(ptr2);
    //ptr1 = ptr2; <- Does not compile!
    ptr2.reset();
    return 0;
}
```

```
~/B-PDG-300> ./a.out
Eccleston is alive
Tennant is alive
Smith is alive
Eccleston is dead
Tennant is touched
Smith is touched
Whittaker is alive
Whittaker is dead
Capaldi is alive
Tennant is dead
Capaldi is dead
Smith is dead
```

Exercise 3 - List

Turn in: `List.hpp/cpp`
Provided files: `IObject.hpp`

In this exercise, you're going to create your own implementation of a list (of `IObject`) in C++! It's time to have a strong and secure container to store our objects.

Create your own `List` class with the following member functions :

- ✓ `bool empty()const` : returns `true` if the list is empty.
- ✓ `std::size_t size()const` : returns the number of elements in the list.
- ✓ `IObject*& front()` : returns the first object in the list.
- ✓ `IObject* front()const` : returns the first object in the list.
- ✓ `IObject*& back()` : returns the last object in the list.
- ✓ `IObject* back()const` : returns the last object in the list.
- ✓ `void pushBack(IObject* obj)` : adds the given `obj` to the back of the list. `obj` can be `nullptr`.
- ✓ `void pushFront(IObject* obj)` : adds the given `obj` to the front of the list. `obj` can be `nullptr`.
- ✓ `void popFront()` : removes the first element of the list.
- ✓ `void popBack()` : removes the last element of the list.
- ✓ `void clear()` : removes every elements from the list.
- ✓ `void forEach(void(*function)(IObject*))` : calls the `function` for every object in the list, from front to back.

A `List` is not copyable or assignable. In case of an invalid operation on a list, such as calling `front` on an empty list, you must throw a `List::InvalidOperationException` that inherits from `std::exception`.

It does not matter *how* your list works, as long as it works. A private `List::Node` nested class to represent a node of the list would be a good idea if you don't know where to start.

Here is a sample main and its expected output:

```cpp
void touch(IObject* object)
{
    if (object != nullptr)
        object->touch();
}

int main(void)
{
    try {
        List list;

        list.pushBack(new TestObject("Kermit"));
        list.pushBack(new TestObject("Miss Piggy"));
        list.pushFront(nullptr);
        list.front() = new TestObject("Fozzie");
        list.pushBack(nullptr);
        list.forEach(touch);
        list.clear();
        list.popBack();
        list.pushFront(new TestObject("Gonzo"));
    } catch (const List::InvalidOperationException& e) {
        std::cout << "Invalid operation on a list" << std::endl;
    }
    return 0;
}
```

Terminal

```
~/B-PDG-300> ./a.out
Kermit is alive
Miss Piggy is alive
Fozzie is alive
Fozzie is touched
Kermit is touched
Miss Piggy is touched
Fozzie is dead
Kermit is dead
Miss Piggy is dead
Invalid operation on a list
```

Exercise 4 - Smart pointers, shared pointers

Turn in : SharedPointer.hpp/cpp
Provided files : IObject.hpp

UniquePointers are cool, but them being not copyable can sometime be... annoying. Let's make smart pointers sharing the same pointer!

A SharedPointer works the same way as a UniquePointer except its pointer can be copied and shared between multiple instances. This can be useful but it's also quite complicated to implement. SharedPointers managing the same pointer share a counter to keep tracks of how many instances of the pointer exist.

Implement the SharedPointer class, deducing its behavior from the following example :

```
int main(void)
{
    SharedPointer ptr1;
    SharedPointer ptr2(new TestObject("O'Neill"));
    SharedPointer ptr3(ptr2);

    ptr1 = ptr3;
    ptr2->touch();
    std::cout << ptr1.use_count() << std::endl;
    ptr1.reset(new TestObject("Carter"));
    std::cout << ptr1.use_count() << std::endl;
    ptr3.swap(ptr1);
    (*ptr3).touch();
    ptr1.reset();
    std::cout << ptr1.use_count() << std::endl;
    ptr2 = new TestObject("Jackson");
    return 0;
}
```

```
~/B-PDG-300> ./a.out
O'Neill is alive
O'Neill is touched
3
Carter is alive
1
Carter is touched
0
Jackson is alive
O'Neill is dead
Carter is dead
Jackson is dead
```

Exercise 5 - List iterators

Turn in : `List.hpp/cpp`
Provided files : `IObject.hpp`

We need to be able to iterate on your list. Using `List::Iterator`, we have a safe tool to inspect our elements, moving from node to node from the begin to the end of the list.

Create a `List::Iterator` nested class with the following operator overloads :

- ✓ `IObject* operator*()const` : returns the object pointer within the current node.
- ✓ `Iterator& operator++()` : advances the iterator to the next node.
- ✓ `bool operator==(const Iterator& it)const` : returns true if the two iterators refers to the same node.
- ✓ `bool operator!=(const Iterator& it)const` : *take a guess.*

A `List::Iterator::OutOfRangeException` inheriting from `std::exception` is thrown when an operation occurs on an invalid `List::Iterator`.

The `List` class must now be improved to take advantage of `List::Iterator`. Add the following member functions to the `List` class :

- ✓ `List::Iterator begin()const` : returns an iterator to the first element of the list.
- ✓ `List::Iterator end()const` : returns an iterator to the end of the list. Any operation on the end iterator will throw a `List::Iterator::OutOfRangeException`.
- ✓ `List::Iterator erase(List::Iterator it)` : removes the element from the list and returns an iterator to the next element.
- ✓ `List::Iterator insert(List::Iterator it, IObject* obj)` : inserts a new element at the given location and returns an iterator to the new element.

An invalid `List::Iterator` given to `erase` or `insert` will result in a `List::InvalidIteratorException`.

Here is a sample main and its expected output :

```
void touch(IObject* object)
{
    if (object != nullptr)
        object->touch();
}

int main(void)
{
    try {
        List list1;

        list1.pushBack(new TestObject("Naruto"));
        list1.pushBack(new TestObject("Sasuke"));
        list1.pushBack(new TestObject("Sakura"));
        list1.pushBack(nullptr);
        list1.pushBack(new TestObject("Serge"));
        for (List::Iterator it = list1.begin(); it != list1.end(); ++it)
            if (*it != nullptr)
                (*it)->touch();
        list1.erase(list1.erase(list1.begin()));
        list1.insert(list1.begin(), new TestObject("Orochimaru"));
        list1.insert(list1.end(), new TestObject("Tsunade"));
        list1.forEach(touch);

        List list2;

        list2.pushFront(new TestObject("Jiraya"));
        list1.erase(list2.begin());
    } catch (const List::Iterator::OutOfRangeException& e) {
        std::cout << "Iterator out of range" << std::endl;
    } catch (const List::InvalidIteratorException& e) {
        std::cout << "Invalid iterator" << std::endl;
    }

    return 0;
}
```

Terminal

```
~/B-PDG-300> ./a.out
Naruto is alive
Sasuke is alive
Sakura is alive
Serge is alive
Naruto is touched
Sasuke is touched
Sakura is touched
Serge is touched
Naruto is dead
Sasuke is dead
Orochimaru is alive
Tsunade is alive
Orochimaru is touched
Sakura is touched
Serge is touched
Tsunade is touched
Jiraya is alive
Jiraya is dead
Orochimaru is dead
Sakura is dead
Serge is dead
Tsunade is dead
Invalid iterator
```

Exercise 6 - Smart pointers, move operators

Turn in : UniquePointer.hpp/cpp
Provided files : IObject.hpp

UniquePointers are very good at managing memory efficiently, but they are quite limited. It would be nice to be able to move the content of a UniquePointer from one to another. Thanks to Bjarne Stroustrup, modern C++ provides such a feature.

For this exercise, you have to implement the move constructor and the move assignment operator of your UniquePointer class. With these improvements, you will be able to run the following code safely :

```
UniquePointer createObject(const std::string &name)
{
    UniquePointer ptr = new TestObject(name);

    return ptr;
}

int main(void)
{
    UniquePointer ptr1 = createObject("Charles de Gaulle");
    UniquePointer ptr2 = createObject("Georges Pompidou");
    //UniquePointer ptr3 = ptr1; <- Does not compile

    {
        UniquePointer tmp(new TestObject("Valery Giscard D'Estaing"));

        //ptr1 = tmp; <- Does not compile
        ptr1 = std::move(tmp);
    }
    ptr2 = UniquePointer(new TestObject("Francois Mitterrand"));
    ptr1.reset(new TestObject("Jacques Chirac"));
    ptr1->touch();
    return 0;
}
```

Terminal

```
~/B-PDG-300> ./a.out
Charles de Gaulle is alive
Georges Pompidou is alive
Valery Giscard D'Estaing is alive
Charles de Gaulle is dead
Francois Mitterrand is alive
Georges Pompidou is dead
Jacques Chirac is alive
Valery Giscard D'Estaing is dead
Jacques Chirac is touched
Francois Mitterrand is dead
Jacques Chirac is dead
```

v 3.2

{EPITECH}