

Travaux Pratiques
Programmation Multi-Paradigme
Licence 3 Informatique

Julien BERNARD

Table des matières

Projet n°1 : Sérialisation dans un fichier	3
Étape 1 : Fichier binaire	3
Étape 2 : Opérateurs de sérialisation	3
Exemple d'utilisation	3
 Projet n°2 : Signals	 5
Étape 1 : Signal	5
Étape 2 : Combineur	5
Exemple d'utilisation	6
 Projet n°3 : Unités et quantités	 7
Étape 1 : Unités	7
Étape 2 : Quantités	7
Exemple d'utilisation	7

Consignes communes à tous les projets

Au cours de cette UE, vous avez **trois** projets à réaliser à raison d'un projet pour deux séances de trois heures de travaux pratiques encadrées. Les projets sont à faire et à rendre dans l'ordre du présent sujet.

Pour chaque projet, vous devrez implémenter une interface donnée dans un fichier d'en-tête, ainsi qu'un ensemble de tests unitaires pour cette interface. Les tests serviront à montrer que votre implémentation est correcte et complète.

Projet n°1 : Sérialisation dans un fichier

Le but de ce projet est d'implémenter des classes pour lire et écrire des fichiers binaires, pour réaliser une bibliothèque de sérialisation. La sérialisation consiste à stocker des données en format binaire portable. Elle est utilisée pour la sauvegarde de fichiers, les protocoles réseaux, etc.

Étape 1 : Fichier binaire

Dans cette étape, vous devrez implémenter deux classes, `OBinaryFile` pour les fichiers en écriture, et `IBinaryFile` pour les fichiers en lecture. Comme il s'agit de classes représentant des ressources, il sera nécessaire d'appliquer la *Rule of Five*.

L'implémentation de ces deux classes utilisera l'API C qui se trouve dans `<cstdio>`.

Étape 2 : Opérateurs de sérialisation

Dans cette étape, vous utiliserez les deux classes définies précédemment pour sérialiser et désérialiser un ensemble de types de base en surchargeant les opérateurs `<<` et `>>`. Attention, le format de sérialisation devra être en *big endian*. On ne prendra pas en charge les pointeurs et les références.

Puis vous aurez à sérialiser des types conteneurs de la bibliothèque standard de manière générique.

Exemple d'utilisation

```
#include <cassert>
#include "Serial.h"

struct Foo {
    int16_t i;
    double d;
    std::string s;
};

serial::OBinaryFile& operator<<(serial::OBinaryFile& file,
    const Foo& foo) {
    return file << foo.i << foo.d << foo.s;
}

serial::IBinaryFile& operator>>(serial::IBinaryFile& file,
    Foo& foo) {
    return file >> foo.i >> foo.d >> foo.s;
}

int main() {
    Foo foo;
    foo.i = 42;
    foo.d = 69.0;
    foo.s = "Hello";
```

```
{
    serial::OBinaryFile out("foo.bin");
    out << foo;
}

struct Foo copy;

{
    serial::IBinaryFile in("foo.bin");
    in >> copy;
}

assert(foo.i == copy.i);
assert(foo.d == copy.d);
assert(foo.s == copy.s);
}
```

Projet n°2 : Signals

Le but de ce projet est d'implémenter une classe de signal. Une classe de signal est chargée de transmettre un signal à un ensemble de fonctions.

Étape 1 : Signal

Concrètement, on enregistre des fonctions de rappel (souvent appelées *slot* ou *callback*) dans le signal, puis on peut émettre un signal qui va transmettre les paramètres à toutes les fonctions de rappel.

La fonction `connectSlot` permet de connecter une fonction de rappel au signal. Cette fonction renvoie un identifiant qui permet de déconnecter la fonction si nécessaire. La fonction `disconnectSlot` permet de déconnecter une fonction de rappel, elle prend en paramètre l'identifiant reçu de la fonction précédente. Enfin, la fonction `emitSignal` émet un signal et appelle toutes les fonctions de rappel connectées. Elle a comme paramètres les mêmes paramètres que les fonctions de rappel.

Il est très vivement conseillé d'étudier la documentation de `std::function` pour bien comprendre comment procéder. En particulier, il sera nécessaire de modifier la manière dont est déclarée la classe `Signal`.

Étape 2 : Combineur

Les fonctions de rappel peuvent avoir un type qui ne soit pas `void`. Dans ce cas, l'utilisateur peut être intéressé à récupérer ces différents résultats et les combiner. C'est pourquoi la classe `Signal` prend en paramètre template une classe `Combiner`.

Un combineur est une classe qui répond à l'interface suivante. Elle définit un type `result_type` qui est le résultat du combineur, c'est-à-dire le résultat des données récupérées des différentes fonctions de rappel une fois combinées. Ensuite, un combineur doit implémenter deux fonctions : premièrement la fonction `combine` qui prend en paramètre un objet de type compatible avec le type de sortie des fonctions de rappel et qui le combine avec les résultats déjà acquis ; deuxièmement une fonction `result` qui envoie le résultat de la combinaison des résultats des fonctions de rappel.

Dans la classe `Signal`, le type `result_type` est strictement identique au type `result_type` du combineur. De plus, le constructeur de `Signal` prend en paramètre un combineur et la fonction `emitSignal` renvoie le résultat du combineur.

Il y a trois combineurs à implémenter :

- `DiscardCombiner` qui va écarter tous les résultats des fonctions de rappels ;
- `LastCombiner` qui ne va conserver que le dernier résultat, c'est-à-dire le résultat de la dernière fonction de rappel enregistrée dans le signal ;
- `VectorCombiner` qui va stocker tous les résultats dans un `std::vector` dans le même ordre que l'enregistrement des fonctions de rappel.

Attention, `result_type` peut être `void`, de même que le type de retour des fonctions de rappel.

Exemple d'utilisation

```
#include <cstdio>
#include "Signal.h"

void callback(int param) {
    std::printf("Hello %i\n", param);
}

int main() {
    // define a signal for functions that takes an int
    // and returns void
    sig::Signal<void(int)> sig;

    // connect a simple function
    sig.connectSlot(callback);

    // connect a lambda function
    int res = 0;
    sig.connectSlot([&res](int x) { res = x; });

    // emit the signal
    sig.emitSignal(1);

    // here res equals 1 and "Hello 1" is printed on stdout
}
```

Projet n°3 : Unités et quantités

Le but de ce projet est d'implémenter une bibliothèque de mesures physiques qui permet de coder les unités du système international avec le système de type de C++.

Étape 1 : Unités

La première partie consiste à définir une classe templée `Unit` représentant une unité. De prime abord, la classe est déjà définie dans le projet et il semble qu'il n'y ait pas beaucoup de choses à faire. En fait, vous vous apercevrez qu'il sera nécessaire d'implémenter des types supplémentaires pour gérer les résultats des opérations. Il est conseillé de placer tous ces types dans un sous-espace de nom `details` puisqu'ils ne font pas partie de l'interface de la bibliothèque.

Il est fortement recommandé de lire la documentation de `std::ratio` dont vous aurez besoin dans la suite et qui peut vous servir d'inspiration pour cette partie du projet.

Étape 2 : Quantités

La seconde partie consiste à implémenter une classe `Qty` représentant des quantités. Les quantités sont représentées par un entier, associé à un type unité et un ratio. Par exemple une quantité représentant des millimètres aura comme type `Qty<Metre, std::milli>`.

La difficulté ici consiste à implémenter correctement les opérateurs de comparaisons et les opérateurs arithmétiques, en prenant en compte les unités mais aussi les ratios.

Exemple d'utilisation

```
#include "Units.h"

int main() {
    using namespace phy::literals;

    auto velocity = 100000_metres / 3600_seconds; // 100 km/h

    phy::Qty<Metre, std::milli> mm(32);
    auto nm = phy::qtyCast<phy::Qty<Metre, std::nano>>(mm);
}
```