IN202 - Programmation Orientée Objet pour l'embarqué
TP1

1 Bonnes pratiques

La programmation devient rapidement une affaire de *communication* avant tout. La part du temps passé à *lire et à comprendre le code* d'un autre ou son propre code peut devenir prédominante. Il est donc plus important d'écrire un code compréhensible qu'un code juste : un code faux mais clair est vite corrigé, un code juste mais incompréhensible provoque des erreurs.

Voici un guide de bonnes pratiques que nous vous conseillons de suivre scrupuleusement. Tout écart de votre part sera à vos risques et périls même s'ils peuvent être pertinents dans certaines situations.

1.1 Organisation du travail

— Chaque exercice doit être effectué au sein d'un nouveau dossier.

— La compilation complète d'un projet doit être possible en un seul clic ou une seule commande simple : utilisez un IDE ou un Makefile [1].

1.2 Conventions de codage

Voici quelques règles d'écriture très répandues :

— Seul le nom d'une classe commence par une majuscule.

— Les noms de varaibles utilisés dans le préprocesseur sont en majuscules (e.g. `#define MAX 42`).

— Les macros comme dans le point précédent doivent être évitées en C++. Préférer les constantes.

— Les noms doivent être aussi explicites que nécessaire mais rester courts. Prendre en compte la portée d'un nom : un nom utilisé dans tout le code doit être très explicite mais une variable locale peut être nommée plus succinctement à l'aide d'abréviations par exemple.

— préfixer les variables membres d'une classe par un `m` ou utiliser le signe distinctif de son choix.

— `int nbVoitures` ou `int nb_voitures` ? Prenez un parti et gardez le.

— Jamais de nombre magique, c'est à dire de valeur littérale tombée du ciel dans le code. Utiliser des constantes à la place.

1.3 Bonnes pratiques diverses

— `cppreference.com` et `stackoverflow.com` sont indispensables.

— La meilleure documentation est le code lui-même mais elle ne suffit jamais.

— Un commentaire ne doit jamais dire ce que le code dit explicitement. Un commentaire donne des informations de haut niveau, pas une explication du langage. Par exemple, ne dites pas qu'une variable constante est constante dans un commentaire. Cela n'apporte rien de plus que le mot clef `const`. Ceci n'est en aucun cas une dispense d'écrire des commentaires lorsque cela est utile, au contraire.

— Une seule classe par header (fichier `.hpp`).

1. Script de compilation classique sous Linux. Sa rédaction peut devenir subtile. Ne vous y aventurez pas si vous n'êtes pas sûr de vous.

— Tout header doit être protégé contre l'inclusion multiple (c.f. include guars).
— Une seule "action" par ligne.
— Indentation cohérente : toujours le même pas et placement cohérent des accolades.
— Sauter des lignes à bon escient : un code trop compact est indigeste mais un code trop aéré dilue l'information.
— Garder les fonctions courtes (<50 lignes environ).
— Tester son code au moins succinctement.

2 Hello world !

Cet exercice permet de découvrir les outils de développement principaux du C++.

2.1 Compilation manuelle, le B-A-BA

Il est essentiel de savoir utiliser un compilateur C++ tel que `g++`. De très nombreux projets industriels d'envergure sont encore à ce jour compilés à l'aide de fichiers `Makefile` et donc à l'aide de commandes de compilation rédigées manuellement. C'est cette méthode qui offre la plus grande souplesse d'utilisation même si elle peut devenir complexe à utiliser.

1. Créer le fichier `hello_world.cpp` et y copier le code donné dans Source 1. Ce code utilise une construction inédite : `cout << ... << ...;`. Cette ligne est équivalente à `printf("Hello world!\n");`. Cette nouvelle construction s'appelle un flux et ne sera pas développée en cours, `printf` étant suffisant.

```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Listing 1: hello_world.cpp

2. Compiler ce code à l'aide de la commande

```
$ g++ hello_world.cpp
```

3. Exécuter le fichier `a.out` qui a été généré. Constater qu'il affiche le message attendu.

La commande `g++` n'est jamais utilisée ainsi en pratique. Voici ses options principales :

— `-o output_name` permet de spécifier le nom du fichier créé par la commande.
— `-c` permet de créer un fichier objet. Il est préférable de compiler séparément chaque fichier source (`*.cpp`) en fichier objet (`*.o`) dans un premier temps et de les linker [2] ensuite.
— `-g` permet d'ajouter des informations de débogue exploitables par un débogueur comme `gdb`.
— `-Wall -Wextra` indiquent à `gpp` de donner d'avantage de messages d'avertissement. Il est recommandé de toujours utiliser ces options et de ne les désactiver qu'en cas de nécessité.
— `-Werror` transforme les `Warnings` en `Errors` qui empêchent la compilation. Il est également recommandé d'activer cette option afin d'apprendre les bonnes pratiques et de détecter certaines étourderies qui arrivent même aux meilleurs. À ne désactiver qu'en cas de nécessité.
— `-std=c++11` active la compatibilité avec C++11.

4. Recompiler le fichier `hello_world.cpp` à l'aide des commandes suivantes :

2. Action de rassembler des fichiers objets pour former un exécutable.

```
g++ -std=c++11 -Wall -Wextra -Werror hello_world.cpp -c -o hello_world.o
g++ hello_world.o -o hello_world
```

La première commande permet de générer un fichier objet à partir d'un fichier source. La seconde commande linke les fichiers objets (un seul ici) pour produire un exécutable.

2.2 Les IDE

Un IDE est un Environnement de Conception Intégré. C'est un logiciel qui rassemble de nombreux outils nécessaires au développement logiciel. Entre autre, on retrouve un éditeur de texte, un compilateur et un débogueur.

Parmi les IDE les plus populaires, on peut citer Microsoft Visual Studio, Eclipse CDT, CLion ou encore QtCreator et `Code::Blocks`. Ce dernier est simple open source et relativement complet et son utilisation basique est décrite ici (`Code::Blocks` 16.01).

1. Ouvrir `Code::Blocks` et cliquer sur *Create a New Project*.
2. Passer le premier message puis choisir *Console application* puis valider.
3. Choisir le langage C++ puis valider.
4. Nommer le projet `helloWorld` par exemple et indiquer son emplacement de sauvegarde puis valider.
5. Vérifier que le compilateur utilisé est bien GCC et que les configurations de debug et de release sont cochées puis terminer. Cela permet ensuite de compiler le projet dans un version optimisée ou adaptée au debogage respectivement.

À ce stade, le volet de gauche devrait contenir l'arborescence du projet.

6. Déplier le dossier *Sources* dans l'arborescence du projet et ouvrir `main.cpp`. Le fichier 1 devrait être présent.
7. Cliquer sur *Build* → *Build and run* pour compiler et lancer le projet.
8. Cliquer juste à droit du numéro de la ligne 7 pour y placer un breakpoint, c'est à dire un point où l'exécution du programme doit s'interrompre lors du débogage.
9. Dans le menu *Build* → *Select Target*, sélectionnez la cible *Debug*.
10. Recompiler le projet en cliquant sur *Build* → *Rebuild*.
11. Lancer le débogage en cliquant sur *Debug* → *Start / Continue*.

Le programme doit alors s'arrêter juste avant d'exécuter la ligne 7. Les outils de debogue classiques sont disponibles dans le menu *Debug* → *Debugging windows*.

Enfin, les options de compilation sont accessibles dans le menu *Project* → *Build options* pour les options spécifiques au projet. Les options globales, quant à elles, sont dans le menu *Settings* → *Compiler*.

12. Cocher l'option *Have g++ follow the C++11 ISO C++ language standard* dans les options de compilation globales. C++ 11 est la version la plus répandue à ce jour dans l'industrie mais C++17 prend de l'importance. Nous nous "limiterons" à C++11 dans ces TD/TP.
13. Cocher les option `-Wall` et `-Wextra`. C'est pour votre bien ;)

Les autres fonctionnalités sont laissées à votre sagacité.

3 Nombres complexes

Les nombres complexes sont déjà implémentés dans la bibliothèque standard. Cet exercice propose de (re)découvrir la programmation OO C++ au travers d'une réimplémentation partielle de la classe `Complex`.

3.1 De la structure ...

1. Dans un fichier ccomplex.cpp, déclarer une structure [3] `CComplex` représentant un nombre complexe sous forme algébrique (de la forme $a + ib$). Les champs seront des `double`.
2. Définir la fonction `CCComplex add(CComplex a, CComplex b)` qui retourne la somme de deux nombres complexes et la tester.

3.2 ... à la classe

3. Déclarer la classe `Complex`
4. Si besoin, déclarer le constructeur et le destructeur de `Complex`
5. Déclarer les *accesseurs de lecture* des parties réelle et imaginaire, c'est à dire des fonctions retournant les valeurs des parties réelle et imaginaire.
6. Définir la méthode `Complex add(Complex other)` qui additionne `other` à l'objet `Complex` appelant et retourne le résultat.
7. Définir une méthode permettant d'imprimer le nombre complexe sur la console sous la forme $a + ib$.

4 File d'attente

Cet exercice permet de manipuler les méthodes et attributs de classe statiques.

Nous cherchons à implémenter un système imaginaire de file d'attente. Chaque personne prend un ticket à son arrivée pour attendre son tour à un guichet. Le passage est effectué suivant le numéro de ticket affiché au guichet.

1. Créer une classe `Ticket` avec les propriétés suivantes :
   - Chaque `Ticket` instancié possède un numéro unique.
   - À chaque instanciation d'un `Ticket`, un compteur indiquant le numéro du prochain `Ticket` est automatiquement incrémenté.
   - Après appel du ticket au guichet, ce dernier doit être détruit. Implémentez un compteur indiquant le `Ticket` suivant appelé au guichet.
   - Chacun des deux compteurs possède un accesseur de lecture. Il doit être utilisable sans disposer d'une instance de la classe `Ticket`.
   - Une méthode retournant un booléen qui indique si le tour d'un `Ticket` est venu.
2. Tester la classe `Ticket`. Afin de causer la destruction d'un objet, vous pouvez l'allouer dynamiquement à l'aide d'un new ou utiliser un bloc entre `{}`. Les variables déclarées dans ce bloc sont détruites à sa sortie. Un tel bloc peut exister sans suivre un `if` ou toute autre structure de contrôle.

3. En C++ une `class` et une `struct` sont identiques si ce n'est que les champs d'une `class` sont privés par défaut là où ceux d'une `struct` sont publics. Cela permet à la syntaxe des `struct` en C de préserver son sens en C++.