TP Programmation C/C++
Fabrication d'un programme simple

Sommaire

Les objectifs de ce tp sont de comprendre et mettre en oeuvre la fabrication d'un programme simple. Beaucoup de conseils sont issus du livre de référence de Bjarne Stroustrup (www.programmation.stroustrup.pearson.fr).

Les Travaux Pratiques ont pour but d'établir ou de renforcer vos compétences pratiques. Vous pouvez penser que vous comprenez tout ce que vous lisez ou tout ce que vous a dit votre enseignant mais la répétition et la pratique sont nécessaires pour développer des compétences en programmation. Ceci est comparable au sport ou à la musique ou à tout autre métier demandant un long entraînement pour acquérir l'habileté nécessaire. Imaginez quelqu'un qui voudrait disputer une compétition dans l'un de ces domaines sans pratique régulière. Vous savez bien quel serait le résultat.

Manipulations

Objectifs

L'objectif de cette partie est la mise en oeuvre de la chaîne de fabrication `gcc/g++` sous GNU/Linux.

Étape n°1 : création de votre espace de travail

Dans votre répertoire personnel, créez un répertoire "dev" où vous stockerez l'ensemble des vos TP de développement. Entrez dans ce répertoire et faites un nouveau répertoire dedans nommé "tp1" où vous stockerez vos travaux pour ce premier TP.

Pour réaliser cela, vous pouvez soit utiliser l'**interface graphique avec l'explorateur de fichiers** (`nautilus` par exemple) soit utiliser une **console** (`Konsole` pour une environnement **KDE** ou `Terminal` pour un environnement **Gnome**) en tapant simplement les commandes suivantes :

```
$ mkdir dev
$ cd dev
$ mkdir tp1
$ cd tp1
```

✍ *La commande* ***mkdir*** *premet de créer un nouveau répertoire et la commande* ***cd*** *de se déplacer à l'intérieur de celui-ci. Le dollar (**$**) représente le prompt ou invite de commandes.*

☞ **L'Annexe n°3 fournit une liste de commandes de base sous GNU/Linux.**

Étape n°2 : édition du programme source

À l'aide d'un éditeur de texte (vi, **vim**, emacs, kwrite, kate, gedit, **geany** sous GNU/Linux ou Notepad, Notepad++, UltraEdit sous Windows, ...), tapez (à la main, pas de copier/coller, histoire de bien le lire et de s'habituer à la syntaxe !) le programme suivant dans un fichier que vous nommerez "`helloworld.cpp`" :

```cpp
#include <iostream>

using namespace std;

int main()
{
   int decompte = 5; // je suis une variable entière initialisée avec la valeur 5

   while(decompte > 0) // je suis une boucle qui fait ... tant que la variable decompte est
       supérieur à 0
   {
      cout << "Hello ";
      cout << "world" << " !" << endl;
      decompte = decompte - 1; // je suis une instruction qui effectue deux opérations : une
          soustraction puis une affectation
   }

   return 0;
}
```

Hello world (version 2) en C++

☞ **Si vous l'utilisez (et c'est fortement conseillé), voir l'Annexe n°2 sur l'éditeur de texte vim.**

Étape n°3 : vérification du bon fonctionnement du compilateur

Tout d'abord ouvrez une console et vérifiez que le compilateur C++ est bien installé en tapant simplement la commande suivante :

```
$ g++
```

Si cela vous répond "`g++: no input file`" ou "`g++: pas de fichier à l'entrée`", alors tout va bien : votre GNU/Linux a bien réussi à exécuter le compilateur... et ce dernier se plaint juste que vous ne lui avez pas dit quoi compiler.

L'installation du compilateur est bien faite et vous pourrez continuer.

Si au contraire GNU/Linux vous répond (en français par exemple) "`bash: g++ : commande introuvable`". Alors c'est que l'interpréteur de commandes (`bash`) n'est pas en mesure de trouver le compilateur installé ou qu'il n'est pas du tout installé. Demandez alors l'aide de l'enseignant.

Étape n°4 : placement dans le bon répertoire

Avant de pouvoir compiler notre premier programme, il nous faut tout d'abord nous déplacer dans la console (la fenêtre noire) pour nous placer dans le repertoire où se trouve le fichier "`helloworld.cpp`" créé précédemment. Pour cela, appprendre quelques commandes `bash` est nécessaire. La première commande à connaitre est "`ls`" ("`ls -l`" pour avoir plus de détails) qui affiche le contenu du répertoire dans lequel vous vous trouvez actuellement. Essayez ! La commande équivalente sous Windows est "`dir`".

La deuxième commande à connaitre est "`cd`" qui change le répertoire où vous vous trouvez : par exemple, pour aller dans le sous-répertoire "dev", vous allez taper :

```
$ cd dev
```

Remarquez que l'invite de commande (en anglais on appelle ça le "prompt", c'est-à-dire le message qui précède le curseur sur la dernière ligne qui vient d'apparaitre) contient toujours le nom du répertoire courrant.

Recommencer l'opération cette fois pour rentrer dans le sous-répertoire "tp1" :

```
$ cd tp1
```

Si vous voulez remonter d'un niveau, rien de plus simple car le nom de répertoire "`..`" indique, où que vous soyez, le répertoire qui se trouve immédiatement au dessus. On l'appelle le **répertoire parent**.

```
$ cd ..
```

Un autre nom de répertoire particulier est "`.`" : c'est le répertoire dans lequel vous êtes actuellement. On l'appelle le **répertoire courant**.

Souvenez-vous que sous Linux, il faut taper "`./helloworld`" pour lancer l'exécution du programme "`helloworld`" (sous Linux, les fichiers exécutables n'ont pas d'extension particulière contrairement à Windows qui possède l'extension "`.exe`" pour les programmes). En fait cela signifie : "dans le répertoire courant" / "le fichier helloworld".

A votre avis, quel est le résultat de la commande suivante :

```
$ cd .
```

Pourquoi ?

Vous avez peut être remarqué qu'une barre oblique sépare les répertoires, et qu'elle n'est pas dans le même sens sous Windows "\" et Linux "/" ? Apprennez à les repérer et à ne pas vous tromper ! Au lieu de faire deux fois la commande "`cd`", on aurait pu aller directement au bon endroit en une seule fois :


```
cd dev\tp1 Sous Windows
cd dev/tp1 Sous Linux
```

Faites maintenant "`ls -l`" (ou "`dir`" et vérifiez que votre fichier "`helloworld.cpp`" apparait bien dans la liste. Si ce n'est pas le cas, appelez l'enseignant à l'aide.

Étape n°5 : fabrication (enfin !) du premier programme

Cela se fait comme vu en cours avec les deux commandes suivantes :

```
g++ -Wall -c helloworld.cpp
```

qui réalise le **"pré-processing", la compilation et l'assemblage** du fichier source "`helloworld.cpp`" en un fichier objet "`helloworld.o`" dans le même repertoire. Faites "`ls -l`" pour vérifier que le fichier "`helloworld.o`" a bien été créé.

Vous devez ensuite faire :

```
g++ -Wall -o helloworld helloworld.o
```

qui réalise l'**édition des liens** entre les divers fichiers ".o" (unification des variables et des fonctions contenues dans ces différents fichiers), puis qui produit le fichier exécutable "`helloworld`".

Vous pouvez maintenant démarer le programme, sous Linux, souvenez-vous qu'il faut indiquer que le programme se trouve dans le repertoire courant en tapant : "./helloworld" pour le démarrer.

```
./helloworld
```

Étape n°6 : analyse des fichiers

Vous avez manipulé différents types de fichiers. En informatique, on distingue essentiellement deux types de fichiers :

- Les fichiers « texte » qui ont un contenu pouvant être interprété comme du texte (une suite de bits représentant un caractère encodé à l'origine en *ASCII*).
- Les fichiers « binaire » qui respectent un format de fichier (convention normalisée ou non) utilisé pour représenter et stocker des données. Tout ce qui n'est pas un fichier texte est un fichier binaire ! Exemples : code machine (exécutable), fichiers multimédias (images, sons, vidéos, traitement de texte, etc.), fichiers archives, fichiers compressés, ...

✍ *Le système GNU/Linux vous donne accès à une documentation riche et variée. Il existe différentes commandes pour accèder à cette documentation :* `help`, `man`, `info` *... Pour l'instant, prenez l'habitude de lire le* ***manuel*** *en utilisant la commande* `man`. *Par exemple :* `man ascii`.

Commençons par utiliser la commande `file` pour déterminer les différents types de fichiers :

➟ **Quel est le type du fichier `helloworld.cpp` ?**

```
$ file helloworld.cpp
helloworld.cpp: UTF-8 Unicode C program text
```

C'est un fichier « texte » (ici endodé en UTF-8 : cf. `man utf-8`).

➟ **Quel est le type du fichier `helloworld.o` ?**

```
$ file helloworld.o
helloworld.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```


C'est un fichier « binaire » (ici le format est ELF « *Executable and Linking Format* » qui correspond à du code machine pour une plate-forme GNU/Linux 64 bits : cf. `man elf`).

➟ **Quel est le type du fichier `helloworld` ?**

```
$ file helloworld
helloworld: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses
    shared libs), for GNU/Linux 2.6.24, ... not stripped
```

C'est un fichier « binaire » (ici un exécutable lié dynamiquement pour une plate-forme GNU/Linux 2.6.24 64 bits).

Petite différence : le `helloworld.o` est un fichier **objet** (`relocatable`) et `helloworld` est un exécutable (`executable`). Les deux fichiers contiennent du code machine.

Pour visualiser le contenu d'un fichier, on va utiliser la commande `hexdump` qui va nous afficher sur 3 colonnes : l'adresse des octets dans le fichier, leurs valeurs en hexadécimal et la traduction en ASCII.

➟ **Que contient le fichier `helloworld.cpp` ?**

```
$ hexdump -C helloworld.cpp
00000000 2f 2f 20 43 65 20 70 72 6f 67 72 61 6d 6d 65 20 |// Ce programme |
00000010 61 66 66 69 63 68 65 20 6c 65 20 6d 65 73 73 61 |affiche le messa|
00000020 67 65 20 22 48 65 6c 6c 6f 20 77 6f 72 6c 64 20 |ge "Hello world |
00000030 21 22 20 c3 a0 20 6c 27 c3 a9 63 72 61 6e 0a 0a |!" .. l'..cran..|
00000040 23 69 6e 63 6c 75 64 65 20 3c 69 6f 73 74 72 65 |#include <iostre|
00000050 61 6d 3e 0a 0a 75 73 69 6e 67 20 6e 61 6d 65 73 |am>..using names|
00000060 70 61 63 65 20 73 74 64 3b 0a 0a 69 6e 74 20 6d |pace std;..int m|
00000070 61 69 6e 28 29 0a 7b 0a 20 20 20 63 6f 75 74 20 |ain().{. cout |
00000080 3c 3c 20 22 48 65 6c 6c 6f 20 77 6f 72 6c 64 20 |<< "Hello world |
00000090 21 5c 6e 22 3b 20 2f 2f 20 41 66 66 69 63 68 65 |!\n"; // Affiche|
000000a0 20 22 48 65 6c 6c 6f 20 77 6f 72 6c 64 20 21 22 | "Hello world !"|
000000b0 0a 20 0a 20 20 20 72 65 74 75 72 6e 20 30 3b 0a |. . return 0;.|
000000c0 7d 0a                                          |}.|
```

Pour visualiser seulement le contenu « texte », on peut utiliser les commandes `cat`, `more`, `strings` ...

```
$ cat helloworld.cpp
// Ce programme affiche le message "Hello world !" à l'écran

#include <iostream>

using namespace std;

int main()
{
   cout << "Hello world !\n"; // Affiche "Hello world !"

   return 0;
}
```

➟ **Que contient le fichier `helloworld.o` ?**

```
$ hexdump -C helloworld.o
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF............|
00000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00 |..>.............|
00000020 00 00 00 00 00 00 00 00 e0 01 00 00 00 00 00 00 |................|
00000030 00 00 00 00 40 00 00 00 00 00 40 00 0f 00 0c 00 |....@.....@.....|
```



```
00000040 55 48 89 e5 be 00 00 00 00 bf 00 00 00 00 e8 00 |UH..............|
00000050 00 00 00 b8 00 00 00 00 5d c3 55 48 89 e5 48 83 |........].UH..H.|
00000060 ec 10 89 7d fc 89 75 f8 83 7d fc 01 75 2a 81 7d |...}..u..}..u*.}|
00000070 f8 ff ff 00 00 75 21 bf 00 00 00 00 e8 00 00 00 |.....u!.........|
00000080 00 b8 00 00 00 00 ba 00 00 00 00 be 00 00 00 00 |................|
00000090 48 89 c7 e8 00 00 00 00 c9 c3 55 48 89 e5 be ff |H.........UH....|
000000a0 ff 00 00 bf 01 00 00 00 e8 ad ff ff ff 5d c3 00 |.............]..|
000000b0 48 65 6c 6c 6f 20 77 6f 72 6c 64 20 21 0a 00 00 |Hello world !...|
000000c0 00 00 00 00 00 00 00 00 00 47 43 43 3a 20 28 55 |.........GCC: (U|
000000d0 62 75 6e 74 75 2f 4c 69 6e 61 72 6f 20 34 2e 36 |buntu/Linaro 4.6|
000000e0 2e 33 2d 31 75 62 75 6e 74 75 35 29 20 34 2e 36 |.3-1ubuntu5) 4.6|
...
```

✍ *Un fichier « binaire » peut contenir du texte ! La commande* **`strings`** *permet d'afficher seulement le contenu texte d'un fichier binaire.*

Le fichier `helloworld.o` contient du code objet, c'est-à-dire du code machine. Le code machine est une suite d'instructions connues et exécutées par le processeur. Ce code machine peut être « désassemblé » :

```
$ objdump -d helloworld.o

helloworld.o:   file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   be 00 00 00 00          mov    $0x0,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   b8 00 00 00 00          mov    $0x0,%eax
  18:   5d                      pop    %rbp
  19:   c3                      retq
...
```

Question 1. Expliquer les 2 colonnes de la ligne `13:` ?

Question 2. Quel est le nom du langage de programmation qui a été obtenu après désassemblage ?

Question 3. Est-il possible à partir d'un code objet de « remonter » jusqu'à un code en langage C ?

Question 4. Donner la commande qui permet de passer d'un code source en langage C à un code source en assembleur ?

➟ Que contient le fichier `helloworld` ?

La « même chose » que le fichier `helloworld.o` : c'est-à-dire du code machine. La seule différence est que le fichier `helloworld` a été lié dynamiquement pour le rendre **exécutable** sur le système d'exploitation GNU/Linux.

Il est possible de visualiser les bibliothèques logicielles dont dépend l'exécutable avec la commande `ldd` :

```
$ ldd helloworld
        linux-vdso.so.1 => (0x00007fffd41ab000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fd0a8f12000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd0a8b51000)
```



```
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fd0a8854000)
/lib64/ld-linux-x86-64.so.2 (0x00007fd0a923b000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fd0a863e000)
```

✍ *Les bibliothèques logicielles à lien dynamique ont l'extension* `.so` *(Shared Object) sous GNU/Linux et* `.dll` *(Dynamic Link Library) sous ©Windows.*

Question 5. À la suite de ces manipulations, lister les noms et rôles des différentes commandes GNU/Linux utilisées ?

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de cette partie.

Question 6. Quelles sont les quatre parties d'une fonction ?

Question 7. Citez une fonction qui doit apparaître dans tout programme C ou C++.

Question 8. Quel est le rôle du compilateur ? Que fait l'éditeur de liens pour votre programme ?

Question 9. À quoi sert la directive `#include` ? Que signifie l'extension `.h` à la fin d'un nom de fichier en C/C++ ?

Question 10. Quelle est la différence entre un fichier source et un fichier objet ?

Question 11. Qu'est-ce qu'un environnement de développement intégré (EDI) et que fait-il pour vous ?

Exercice 1 : à vous de jouer

À partir des manipulations précédentes, réaliser la fabrication du programme "Hello world" en C :

```c
// Ce programme affiche le message "Hello world !" à l'écran

#include <stdio.h> /* pour printf */

int main()
{
   printf("Hello world !\n"); // Affiche "Hello world !"

   return 0;
}
```

Hello world (version 1) en C

Question 12. Quelle est l'extension à donner au fichier source ?

Question 13. Donner les différentes commandes permettant la fabrication de l'exécutable ?

Question 14. Pourrait-on utiliser le compilateur C++ pour fabriquer l'exécutable ? Pourrait-on utiliser le compilateur C pour fabriquer l'exécutable à partir du code source `helloworld.cpp` ?

Exercice 2 : corriger des erreurs

L'objectif de cet exercice est d'apprendre à interpréter les erreurs signalées par le compilateur.

Question 15. Éditez le fichier "`a-corriger.cpp`" ci-dessous et tentez de le compiler. Quel compilateur faut-il utiliser ? `g++` ou `gcc` ?

```
/Qu'est censé faire ce programme ?

// auteur : e. remy

include <iostrem>

using namespace std;

integer main()
{
 /* Vous devez corriger ce programme et arriver à le compiler puis l'exécuter.
 cout << 'Le programme marche !' << end;
 int valeur = 10
 cout << "valeur =" << valeur << end;
 // Attention : la division de deux entiers est une division euclidienne,
 // c'est-à-dire une division ***ENTIERE*** !
 int quotient = 10 / 3
 cout << "quotient=" << quotient << end;
 int reste = 10 % 3
 cout << "reste=" << reste << end;
 // Si vous voulez faire une division réelle, il faut convertir un des
 // arguments en réel :
 out << "quotient reel =" << valeur / 3.0 <<end; // Cette fois-ci 3.0 est réel
 out << "Fin du programme;
 return 0;
}
```

Un fichier source truffé d'erreurs !

La compilation échoue car le fichier est truffé d'erreurs !

Question 16. Corrigez-les jusqu'à obtenir le bon fonctionnement du programme. Puis, ajouter le commentaire classique au début du programme source.

Quelques consignes importantes :
- Vous pouvez commencer par tenter de corriger toutes les erreurs que vous trouver par vous-même en lisant le programme... mais au delà, c'est au compilateur de vous dire où sont les erreurs de syntaxe.
- Ne considérez que la première erreur signalée par le compilateur : les suivantes peuvent être une conséquence de la mauvaise compréhension de la suite du programme par le compilateur à cause de cette première erreur... il faut donc la corriger en premier ! Une fois qu'elle est corrigée, essayez de recompiler pour voir si vous avez bien corrigé, et s'il reste d'autres erreurs... La programmation est une véritable école de patience !
- Utilisez le numéro de ligne indiqué par le compilateur. Soit l'erreur se trouve à la ligne indiquée... soit un peu avant : le numéro de ligne indiqué est l'endroit où il devient manifeste pour le compilateur que le programme est erroné... mais des fois vous avez pu écrire une bêtise qui n'est pas litéralement fausse et donc que le compilateur accepte pendant quelques lignes... jusqu'à ce que cela devienne clair qu'il y a un problème !

– Si vous ne trouvez pas la source de l'erreur, n'hésitez pas à appeler à l'aide ! L'enseignant est là pour vous aider. Plus tard, quand vous rédigerez vos propres programmes, sachez également que quand on a "le nez dedans", on ne voit pas toujours ses propres fautes, mais qu'elles sont souvent évidentes pour quelqu'un qui a un regard neuf sur votre programme : demander une relecture à un de vos camarades s'avère donc souvent très efficace.

Exercice 3 : faire évoluer un programme

L'objectif de cet exercice est de faire évoluer un programme existant. *C'est une pratique que l'on détaillera par la suite.*

Question 17. Testez le programme ci-dessous. Indiquer (comme vous l'avez appris) ce que fait ce programme ?

```
//

#include <stdio.h>

int main (int argc, char **argv)
{
   int n;

   printf("Donnez un entier : ");
   scanf("%d", &n); // le & signifie l'adresse de la variable n et, c'est nécessaire ici
      pour que la fonction puisse modifier la variable passée en argument
   printf("Vous avez donné l'entier : %d\n", n);

   return 0;
}
```

Une saisie clavier

✍ *Le* `%d` *est un spécificateur de conversion de l'argument* `n`*, un* `int` *ici, qui sera converti en un chiffre décimal signé. La fonction* `scanf()` *lit une donnée depuis le flux d'entrée standard* `stdin` *et l'affecte à la variable* `n`*. Ces fonctions sont décrites dans le chapitre 3 du manuel :* `man 3 printf`*.*

Question 18. Quel est le nom donné au chapitre 3 du manuel ? Quel est la valeur de retour de la fonction `printf` ? `scanf` ?

Question 19. Écrire la version C++ de ce programme en utilisant `cout` (à la place de `printf`) et `cin` (à la place de `scanf`). Que permet de faire `cin` ?

Question 20. Modifiez le programme précédent pour qu'il puisse afficher **n** fois le message "Hello world !" (n étant une valeur saisie par l'utilisateur). Que se passe-t-il si l'utilisateur saisit une valeur négative ?

Bilan

Qu'y a-t-il de si important à propos du programme "Hello world !" ? Son objectif était de vous familiariser avec les outils de base utilisés en programmation.

Retenez cette règle : il faut toujours prendre un exemple extrêmement simple (comme "Hello world") à chaque fois que l'on découvre un nouvel outil. Cela permet de diviser l'apprentissage en deux parties : on

commence par apprendre le fonctionnement de base de nos outils avec un programme élémentaire puis on peut passer à des programmes plus compliqués sans être distraits par ces outils. Découvrir les outils et le langage simultanément est beaucoup plus difficile que de le faire un après l'autre.

Conclusion : cette approche consistant à simplifier l'apprentissage d'une tâche complexe en la décomposant en une suite d'étapes plus petites (et donc plus faicles à gérer) ne s'applique pas uniquement à la programmation et aux ordinateurs. Elle est courante et utile dans la plupart des domaines de l'existence, notamment dans ceux qui impliquent une compétence pratique.

Descartes (mathématicien, physicien et philosophe français) dans le *Discours de la méthode* :

> « diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre. »
>
> « conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu comme par degrés jusques à la connaissance des plus composés ... »

