Jusqu'à présent, vous avez utilisé les commandes "runC" et "runT" afin d'exécuter et tester vos programmes. Comme nous vous l'avions indiqué, ces commandes n'existent pas normalement, nous les avions ajoutées afin de simplifier votre découverte de la ligne de commande sous Linux. Dans ce cours, nous allons vous apprendre les "véritables" commandes qui permettent de compiler/exécuter/tester un programme.

Rendez vous dans le sous-dossier "tasks/hello_world"

Compiler un programme

Pour compiler le code source et obtenir un code binaire executable, lancez la commande suivante :

```
$ g++ -o prog prog.cpp
```

Si vous affichez la liste des fichiers du dossier, vous verrez un fichier nommé "prog". Il s'agit de l'exécutable, du code binaire que nous allons pouvoir exécuter.

Dans cette ligne de commande,

- `g++` est le nom du compilateur pour le langage C++.
- `-o prog` indique le nom (ici "prog") du fichier exécutable qu'on souhaite créer.
- `prog.cpp` est le nom du fichier contenant le code-source.

On peut donc la lire ainsi : "compiler le code-source du fichier prog.cpp et créer l'exécutable prog".

Exécuter un programme

Pour exécuter le programme, lancez la commande suivante :

```
$ ./prog
Hello world!
```

Si vous changez le code source, faites attention de bien toujours le recompiler avant d'exécuter le programme, sinon le code binaire ne change pas !

Commande de compilation plus avancée

En général la commande de compilation est un peu plus complexe :

```
$ g++ -O2 -Wall -Wextra -o prog prog.cpp
```

où

- `-O2` est une option indiquant au compilateur qu'il faut optimiser (rendre plus rapide) le code binaire produit.
- `-Wall` et `-Wextra` indiquent au compilateur d'émettre des avertissements si certaines choses sont détectées dans le code.

L'option d'optimisation est importante car elle améliorera grandement le temps de calcul de votre programme, parfois d'un facteur 5 à 10. Vous pourrez ainsi l'exécuter rapidement sur de gros tests.

Les options pour les avertissements sont très importantes car elle permettent de détecter des bugs potentiels dans votre programme dès la compilation ! Nous avons indiqué ci-dessus les deux options les plus importantes/courantes mais il en existe de nombreuses autres. A titre d'information, la liste

complète des options pour les avertissements est disponible ici mais vous n'avez à priori pas besoin d'autres options que ces deux là.

Redirections de fichiers

Rendez vous dans le sous-dossier "tasks/double" et compilez le code source.

Un programme lit ses données sur ce qu'on appelle l'entrée standard et écrit sur la sortie standard. Il est possible de prendre un fichier (par exemple un fichier test) et l'envoyer sur l'entrée standard de votre programme :

```
$ ./prog < test.01.in
10
```

Cette commande a pris les données contenues dans le fichier "test.01.in" et les a envoyées au programme pour que celui-ci puisse les lire.

La sortie du programme s'affiche dans le terminal. Si la sortie est longue, ou que l'on souhaite la regarder plus tard, il est plus pratique de rediriger la sortie standard du programme vers un fichier, plutôt que directement vers le terminal :

```
$ ./prog < test.01.in > out
$ cat out
10
```

Le fichier "out" contient donc la sortie du programme et rien ne s'affiche dans le terminal.

Comparer deux fichiers

Lorsque la sortie d'un programme est très longue, ou que l'on souhaite automatiser les choses, il devient difficile de comparer manuellement cette sortie à la sortie qu'on souhaitait avoir (le fichier ".out"). Il existe une commande permettant de comparer deux fichiers :

```
$ diff out test.01.out
```

Ici rien ne s'affiche car les deux fichiers sont identiques. Modifiez le fichier "test.01.out" et relancez la commande précédente pour voir ce qui se passe.

Il existe plusieurs options utiles pour "diff" :

- Si vous voulez juste savoir si les fichiers sont égaux ou non et ne pas visualiser tous les détails, utilisez l'option "-q" (q comme quiet) :

  ```
  $ diff -q out test.01.out
  ```

- Si vous souhaitez ignorer les différences qui sont simplement des caractères d'espacement comme ' ', '\t', '\n', utilisez l'option "-w" (w comme white-space) :

  ```
  $ diff -w out test.01.out
  ```

Combiner les commandes

Il est donc nécessaire d'exécuter trois commandes

```
$ g++ -O2 -Wall -Wextra -o prog prog.cpp
$ ./prog < test.01.in > out
$ diff out test.01.out
```

Comme cela est un peu long, nous allons les combiner, de telle sorte qu'une commande s'exécute uniquement si la précédente c'est bien passée (pas d'erreur signalée) :

```
$ g++ -O2 -Wall -Wextra -o prog prog.cpp && ./prog < test.01.in > out && diff out te
```

On peut également éviter la création du fichier intermédiaire "out" et faire directement la comparaison entre la sortie du programme et la sortie de référence :

```
$ g++ -O2 -Wall -Wextra -o prog prog.cpp && diff <(./prog < test.01.in) test.01.out
```

Attention à ne pas mettre d'espace entre le `<` et le `(` sinon cela ne fonctionnera pas. Ce `<( )` exécute la commande située au milieu et envoie le résultat dans un fichier temporaire qu'il "donne" ensuite à la commande diff. Ainsi, au lieu de faire nous-même un fichier temporaire "out", on laisse l'ordinateur s'en charger !

Voici une variante possible :

```
$ g++ -O2 -Wall -Wextra -o prog prog.cpp && ./prog < test.01.in | diff - test.01.out
```

Le "-" comme premier argument du "diff" indique qu'il faut comparer son entrée standard (et non pas un fichier) à son second argument. Le pipe ("|") permet justement d'envoyer la sortie du programme vers l'entrée standard du "diff".

A voir de choisir la variante que vous préférez.

Combiner plusieurs tests

La commande précédente permet de vérifier le résultat d'un programme sur un fichier test, comment faire si on a plusieurs fichiers tests ? On va faire une boucle !

Tout d'abord une boucle pour afficher le nom de tous les fichiers ".in" du répertoire :

```
$ for f in *.in; do echo "$f"; done
test.01.in
test.02.in
test.03.in
```

On fait une boucle sur la variable "f" qui vaudra, tout-à-tour, le nom de chacun des fichiers ".in". On demande ensuite à afficher le contenu de cette variable à l'aide de la commande "echo" et en mettant un "$" devant le nom de la variable.

Si on combine cela avec la commande de compilation/test :

```
$ g++ -O2 -Wall -Wextra -o prog prog.cpp && for f in *.in; do echo "$f" ; diff <(./p
```

Le ";" permet de séparer des commandes indépendantes qui s'exécutent même si la précédente à renvoyée une erreur. Quand au `${f/.in/.out}` il nous permet de prendre le contenu de la variable "f", de remplacer le ".in" par un ".out" et de renvoyer le résultat.

Mesurer le temps d'exécution

Pour mesurer le temps mis par un programme pour s'exécuter, on va utiliser la commande `time` :

```
$ time ./prog < test.01.in
real  0m0.002s
user  0m0.000s
sys   0m0.000s
```

Notez que les temps peuvent être différents sur votre ordinateur.

- Le temps `real` correspond au temps qui s'est effectivement écoulé, dans le monde réel, entre le début et la fin de l'exécution de votre programme. Il peut dépendre de nombreux facteurs, comme par exemple les autres programmes exécutés au même moment par la machine.
- Le temps `user` correspond au temps que le processeur a passé à exécuter les instructions de votre programme, sans compter les appels systèmes.
- Le temps `sys` correspond au temps que le processeur a passé à exécuter les appels systèmes, c'est-à-dire une partie du temps utilisé par des fonctions systèmes pour lire/écrire des données.

Le temps qui vous intéresse est le total du temps `user` et du temps `sys`.

N'oubliez pas que votre machine n'est pas nécessairement du même type que celle sur laquelle votre programme sera évalué, et donc le temps d'exécution du programme sur votre machine ne sera pas le même que le temps d'exécution sur le serveur. Nous vous invitons à faire quelques tests pour établir une correspondance.

Mettre en mémoire la commande de compilation

Il est possible de créer un alias des commandes qu'on utilise. Ouvrez le fichier .bashrc situé à la racine de votre compte

```
$ gedit ~/.bashrc
```

Ajouter à la fin la ligne suivante puis enregistrez le fichier, fermez le terminal et ouvrez en un nouveau.

```
alias g++std='g++ -O2 -Wall -Wextra'
```

Désormais vous pouvez utiliser la commande simplifiée

```
g++std -o prog prog.cpp
```

```
gcjstd -o prog prog.java
```

Au moment de la taper, penser à la touche tabulation pour ne pas retaper tout le "g++std" "gcjstd" vous-même.

Limiter la mémoire utilisée

On appelle "shell" le programme qui, dans le terminal, interprète les différentes commandes que vous tapez. Dès que vous ouvrez le terminal, le shell est déjà lancé et prêt à interpréter vos commandes !

Pour vérifier que votre programme n'utilise pas trop de mémoire, sous Linux, vous pouvez utiliser la commande bash `ulimit`, qui permet de fixer une limite pour certaines ressources. Ces limites seront appliquées à tout programme lancé à partir de ce shell.

Attention : une fois que vous avez fixé une limite, vous ne pouvez pas la réaugmenter sans quitter le shell. Lancez donc un nouveau shell temporairement (commande `bash`) pour faire votre test. Pour

quitter ce nouveau shell, utilisez le raccourci clavier Ctrl+D (touche "Ctrl" et touche "D"). Une fois dans ce nouveau shell vous pouvez utiliser les commandes suivantes.

Pour fixer une limite globale de 16Mo à l'utilisation de la mémoire :

```
$ ulimit -v 16000
```

Vous pouvez remplacer la valeur 16000 par la valeur de votre choix, à exprimer en kilo-octets.

Si vous exécutez ensuite votre programme dans ce shell, une erreur sera affichée si la mémoire est insuffisante. Il peut s'agir d'un simple message "Killed", ou d'une erreur d'allocation, de segmentation....

Dans certains cas, on peut vous indiquer une limite séparée pour la taille de la pile. Vous pouvez la fixer avec l'option -s :

```
$ ulimit -s 10000
```

Encore une fois, la limite est à exprimer en kilo-octets.

Pour combiner les deux commandes :

```
$ ulimit -v 16000 -s 10000
```

A tout moment, vous pouvez afficher les limites actuelles du shell en appelant `ulimit -v` ou `ulimit -s` sans indiquer la taille. Pour plus de détails, utilisez la commande `ulimit -a` qui permet d'afficher toutes les limites qu'il est possible de fixer.

Conclusion

Nous vous avons donné les outils de base pour pouvoir compiler/exécuter/tester vos programmes sous Linux. Il n'est pas nécessaire d'en connaître plus pour résoudre tous les exercices que vous rencontrerez. Cela peut cependant vous aider, à vous de découvrir Linux par vous-même.

Pour les curieux n'hésitez pas à aller voir, dans quelques temps, comment fonctionnent les exécutables runC et runT :

```
$ less ~/bin/runT
$ less ~/bin/runC
```

Ils utilisent des choses non expliquées dans ces petits cours aussi il est normal que vous ne compreniez pas tout. Vous n'avez absolument pas besoin de savoir faire cela, nous vous les indiquons uniquement à titre indicatif.