CHRISTOPHE DEROULERS

SAMUEL CAZAYUS-CLAVERIE

RUDIMENTS INFORMATIQUES À L'USAGE DE LA PHYSIQUE NUMÉRIQUE

UNIVERSITÉ PARIS-SACLAY 2018

Table des matières

1
Découverte d'UNIX

1.1 Introduction

Dans ce chapitre, nous présentons une alternative au maniement de l'ordinateur en interface graphique : la saisie de commandes dans une console. Dans l'usage que nous en ferons, cet outil permet d'exécuter des programmes qui se trouvent sur l'ordinateur, de gérer des fichiers, et de compiler du code source.

Nous nous intéressons aux systèmes fondés sur UNIX tels que Mac OS ou Linux [1]. L'interprète de commande [2] que nous utiliserons se nomme bash ("Bourne-again shell").

1. Les possesseurs de Windows peuvent cependant installer un interprète de commandes bash pour suivre ce tutoriel. Ils peuvent aussi installer Linux sur leur PC, les enseignants seront ravis de les accompagner dans cette conversion au logiciel libre !
2. Génériquement appellé *shell*

1.2 Démarrage de l'interprète

Sur Mac OS, vous pouvez démarrer la console en cherchant l'application "Terminal". Sur Linux, le raccourci clavier ctrl + Alt + T vous permet d'en faire de même. Une fenêtre s'ouvre alors, une instance de bash s'exécutant à l'intérieur. Il vous est alors possible de taper du texte, mais que taper ?

1.2.1 Exécution d'un programme

Si vous souhaitez exécuter un programme présent sur votre ordinateur, il vous suffit de taper son nom ! Par exemple, le programme qui permet de savoir où vous vous situez par rapport à la racine de votre système de fichiers se nomme "pwd" [3], pour savoir où vous vous trouvez, tapez la commande suivante :

3. Print Working Directory

```
pwd
```

L'exécution de certains programmes peut requérir un argument, qu'il nous suffit d'entrer après le nom du programme. C'est le cas de la commande "man" qui permetde consulter la notice d'un programme dont le nom est donné en argument. Exemple :

```
man man
```

Nous découvrons que l'exécution du programme "man" peut être personnalisée au moyen d'options, entrées à la suite d'un trait d'union.

1.2.2 Navigation dans le système de fichiers

Le système de fichiers possède une structure d'arbre. La racine "/" compte parmi ses enfants des répertoires, qui sont des noeuds de l'arbre à l'intérieur desquels nous pouvons nous aventurer. Les enfants qui ne sont pas des répertoires sont des feuilles de l'arbre : c'est le cas des exécutables et des fichiers de texte par exemple.

Le programme qui permet de connaître le contenu du dossier courant est "ls" [4]. Elle peut aussi prendre l'adresse d'un répertoire en argument afin d'en afficher le contenu. La commande :

4. List Segments

```
ls ~
```

Affiche le contenu du répertoire de l'utilisateur, " ~ " étant un raccourci pour désigner son adresse. N'hésitez pas à consulter le manuel pour plus d'informations!

Pour vous déplacer dans l'arborescence vous pouvez utiliser la commande "cd" [5]. *L'autocomplétion* est un outil précieux, elle permet d'entrer le début d'une commande, et de la compléter automatiquement au moyen de la touche de tabulation. Si plusieurs choix sont possibles ils seront affichés à l'écran. Entrez les commandes suivantes

5. Change Directory

```
cd ~/Desktop
pwd
cd ..
pwd
cd Desktop
```

Nous sommes entrés dans le bureau, nous avons vérifié que nous y étions, nous sommes remontés au dossier parent au moyen de "..", nous avons vérifié l'adresse, pour enfin revenir au bureau ...

A présent nous pouvons créer un répertoire au moyen de la commande "mkdir" [6]. Créons un répertoire à notre nom :

6. MaKe DIRectory

```
mkdir SamuelCazayus
```

Pourtant, nous voilà insatisfaits de ce nom que nous souhaiterions le modifier aux noms du binôme, nous pouvons donc renommer le répertoire au moyen de la commande "mv" [7] :

7. MoVe, cette commande sert aussi à déplacer des fichiers. Après tout, renommer ce n'est rien de moins que déplacer sur place!

```
mv SamuelCazayus CazayusDeroulers
```

Entrons à présent dans le répertoire, et créons-y plusieurs sous-répertoires :

```
cd CazayusDeroulers
mkdir codeSource inputData outputData dossierInutile
```

Nous voudrions supprimer dossierInutile, rien de plus simple ! Effaçons-le au moyen de la commande "rm" [8] !

8. ReMove

```
rm -r dossierInutile
```

Nous avons utilisé l'option "-r" de la commande "rm" qui effectue un effacement récursif du dossier et de son contenu, une erreur surviendrait sinon ! Poussons le vice un peu plus loin :

```
mkdir dossierInutile1 dossierInutile2 dossierInutile3
```

Vous pouvez alors appliquer l'effacement en utilisant la *reconnaissance de motif* dans les noms au moyen de l'étoile "*" :

```
rm -rf dossier*
```

Enfin, si un jour vous désiriez passer outre les confirmations d'effacement intempestives, vous pouvez utiliser l'option de forçage :

```
rm -rf dossierInutile
```

Méfiez-vous cependant, l'opération est irréversible, ne dérapez pas !

1.2.3 Exécutables locaux

Certains fichiers présents sur votre ordinateur sont exécutables. Il suffit de saisir un chemin vers un exécutable pour le lancer. Par exemple, la syntaxe d'exécution du programme "monProgramme.o" dans le répertoire courant est "./monProgramme.o". Nous verrons des exemples dans la partie programmation.

Sachez que bash recherche aussi les noms d'exécutables dans les répertoires de la *variable d'environnement* $PATH [9]. Cette variable est une liste de répertoires autorisés à l'exécution : le dossier courant n'y figure pas, d'où la nécessité du "./". Vous pouvez visualiser [10] cette liste en tapant :

9. La syntaxe $NOM est dédiée aux variables d'environnement.

10. Notez que le séparateur de liste est " :".

```
echo $PATH
```

C'est grâce à cette variable d'environnement que vous ne devez pas saisir l'adresse complète de "pwd" ou "man" pour les exécuter. Bien sûr, tous les fichiers ne sont pas exécutables, ce que votre ordinateur sait grâce aux droits d'accès aux fichiers.

1.2.4 Droits d'accès

Chaque fichier sur votre machine possède des droits d'accès en écriture, en lecture et en exécution. Ces droits peuvent être différents pour le propriétaire du fichier, le groupe auquel il appartient, et les autres individus. Pour lire les droits d'accès à vos fichiers, tapez dans l'interprète bash :

```
ls -l
```

Vous obtenez une liste de 10 symbôles. Le premier, -, d ou l, vous indique la nature fichier, répertoire ou lien [11]. Les neuf suivants "wxr wxr wxr" vous indiquent les droits d'accès pour les trois groupes susmentionnés.

L'administrateur système (et lui seul) peut modifier le propriétaire d'un fichier et son groupe au moyen des commandes "chown" [12] et "chgrp" [13]. Il est aussi possible de changer les droits d'accès de ces derniers grâce à la commande "chmod" [14]. Un + ajoute des droits, un - en retire :

```
chmod u+rwx,g+rx-w,o+r-wx file
```

11. Un lien est l'analogue du raccourci windows. Il existe sous UNIX des liens physiques et des liens symboliques.

12. Change Owner

13. Change Group

14. Change Mode

1.2.5 Astuces utiles

1.2.6 Redirection des commandes vers un fichier de texte

La plupart des programmes pensés pour linux effectuent une unique tâche simple. La commande "echo" par exemple renvoie un texte sur la sortie standard de votre programme, et c'est tout ! Cependant, il peut être nécessaire d'effectuer une tâche composée. C'est à cette fin que l'on peut rediriger la sortie d'un programme.

Par défaut, la sortie d'un programme s'effectue sur la sortie standard, vouée à être affichée sur le terminal, par exemple le résultat de la commande

```
echo 'Hello'
```

sera affiché dans le terminal. Certaines sorties, appelées sorties d'erreur sontenvoyées sur le canal d'erreur [15], qui est -lui aussi- affiché par défaut sur le terminal.

Il est cependant possible de rediriger ces sorties dans un endroit plus opportun. Pour rediriger la sortie d'un programme vers un fichier "test.txt", on peut employer :

```
echo 'Hello' > "test.txt"
echo 'Hello' >> "test.txt"
```

Le chevron simple écrase le contenu du fichier avant d'écrire, le chevron double écrit à la fin du fichier.

Si vous souhaitez rediriger autre chose que le canal standard dans un fichier, vous le pouvez : par défaut, la sortie standard porte le numéro 1, et la sortie d'erreur standard le numéro 2 :

15. Par exemple lorsqu'on utilise le flux "std : :cerr" en C++

```
echo 'Hello' 2>"test.txt"
```

Le fichier obtenu est vide, car la commande echo n'a rien envoyé sur la sortie d'erreur. La sortie standard 1 quant-à-elle n'a pas été affectée par le processus et s'affiche dans le terminal.

Il n'y a en fait pas de limite au nombre de redirections, on peut donc d'abord rediriger la sortie standard vers la sortie d'erreur [16], puis la sortie d'erreur vers un fichier texte :

16. Notez la présence de l'esperluette pour désigner le canal d'erreur, sinon vous allez créer un fichier "2" pour réceptionner la sortie standard.

```
echo 'Hello' 1>&2 2>"test.txt"
```

pour en arriver au même résultat qu'avec la redirection simple.

Vous pouvez aussi ignorer la sortie standard au moyen du périphérique "/dev/null"

```
echo 'Hello' 1> /dev/null
```

Rediriger le canal d'erreur permet d'analyser de façon plus lisible les sorties d'erreur d'un programme, par exemple lors de la compilation d'un exécutable C++.

1.2.7 Redirection vers un autre programme

Il est aussi possible de rediriger la sortie d'un programme dans l'entrée d'un autre. C'est l'opérateur "|" qui permet cette opération. Ici, nous combinons la sortie d'echo au filtre "grep". Cette stratégie permet de ne sortir que les lignes comportant une chaîne de caractère [17] donnée.

17. en fait, grep accepte aussi les expressions régulières, mais ce serait l'objet d'un cours à part entière...

```
echo 'Hello' | grep -e "A"
echo 'Hello' | grep -e "l"
```

Bien sûr, c'est le fait de pouvoir rediriger en chaîne qui fait toute la saveur de la fonctionnalité!

Vous voilà passés maîtres dans l'art de l'interprète bash, félicitations! Efforcez-vous de travailler autant que possible en console, vous verrez que l'on y prend vite goût, et que l'on gagne bien souvent du temps!

2
Un peu de C++

2.1 Introduction

Le processeur de votre ordinateur est capable de réaliser très rapidement des opérations entre nombres binaires, la mémoire de votre ordinateur est encodée en binaire et les câbles de votre ordinateur transmettent des messages binaires. Pourtant, votre expérience d'utilisateur est tout autre : votre écran affiche des images, et vous entrez des chiffres et des lettres à l'aide de votre clavier. Les informaticiens ont rendu cela possible grâce à des langages informatiques. Les premiers langages informatiques étaient de bas niveau, c'est-à-dire qu'il comportaient seulement des fonctionnalités proches du langage de l'ordinateur. Par exemple des fonctionnalités permettant l'allocation de mémoire par adresse.

Très vite, des langages de plus haut niveau sont apparus, on peut par exemple citer Mathematica, un logiciel commercial capable d'effectuer du calcul formel. On peut directement y diagonaliser des matrices et y calculer des intégrales complexes.

La notion de langage apporte avec elle la notion de traduction. Comme pour les langages humains, il existe des interprètes, qui traduisent au fûr-et-à-mesure, et des traducteurs qui traduisent l'intégralité d'un texte et la retranscrivent par écrit. Une traduction accélère notablement la saisie du sens par l'utilisateur étranger ; une interprétation ne nécessite pas d'attendre que le livre soit achevé. Certains langages sont interprétés, tels que le Python : l'auteur du code inscrit des instructions dans l'interprète qui les traduit en binaire. D'autres langages sont traduits de fichiers textes en exécutables binaires, c'est le cas du C++. Notons qu'il est possible de compiler des langages qui s'interprètent, mais pas l'inverse ; cependant, en faisant cela, on perd un peu du sel d'un langage interprété. Bien sûr, les interprètes et les compilateurs ne sont pas des personnes physiques mais des programmes. Généralement, ils sont écrits dans un langage de plus bas niveau que celui qu'ils traduisent !

Dans ce chapitre, on se propose de réaliser quelques programmes simples en C++. Comme son nom le suggère, le langage C++ est une version amélio-

rée du langage C. En pratique, nous ne nous soucierons pas de ce qui fait la spécificité du C++ par rapport à son ancêtre. Nous prenons le parti de toujours utiliser la solution la plus élégante lorsque le choix est à possible.

2.2 Concepts liés à la compilation en C++

2.2.1 Compilateur C++

Un compilateur C++ est un programme qui reçoit un fichier texte en entrée, et renvoie un exécutable en sortie. Pour exécuter le programme, il suffit de taper son adresse relativement au chemin renvoyé par la commande pwd dans le shell. On l'utilise alors en bash comme n'importe quel autre programme.

Plusieurs compilateurs de C++ existent : les compilateurs g++ et gcc sous linux, clang sous Mac Os. Il existe aussi des compilateurs commerciaux réalisés par certaines entreprises dans le but d'optimiser l'exécutable binaire pour les processeurs de la marque : bien souvent, la vitesse d'exécution s'en trouve améliorée.

2.2.2 Premier programme

Pour commencer à écrire un programme, placez-vous dans votre dossier codeSource. Nous allons utiliser un éditeur de texte intelligent. Les puristes utiliseront Emacs ou Vim ; nous utiliserons Ntom, présent sur les machines du LAL. Ces éditeurs de texte ont en commun de proposer une coloration syntaxique et une gestion intelligente de l'indentation du programme. De plus ils sont hautement personnalisables : ils constituent votre environnement de travail, choisissez celui que vous préférez ! Vim requiert un peu plus d'érudition qu'Emacs, qui lui-même en requiert plus qu'Atom.

Créons notre premier fichier C++ grâce à Atom. L'extension .cpp est utile pour qu'Atom active une coloration syntaxique pour le C++. L'esperluette permet à l'utilisateur de récupérer la main sur l'exécution sans attendre la fin d'exécution d'Atom.

```
atom helloworld.cpp &
```

Un programme est constitué d'une suite d'instructions. Ces instructions sont lues les unes après les autres. Le caractère de fin d'instruction est un point-virgule ";". Ce dernier est obligatoire.

Saisissez les lignes suivantes :

```
#include <iostream>

int main ()
{
  std::cout << "hello world !" << std::endl;
```

```
    return 0;
}
```

Démistifions-en le sens : la ligne qui commence par un # est une *directive du préprocesseur*. Un terme barbare pour dire qu'elle s'adresse au compilateur, et lui demande d'incorporer la bibliothèque "iostream" [1] au programme.

Par la suite, nous avons affaire à la fonction principale du programme : en C++, une fonction principale, nommée main, doit être présente dans tout programme. Cette fonction doit renvoyer un entier grâce à l'instruction return. C'est la fonction par laquelle commence l'exécution de votre programme.

Dans le corps de la fonction, nous demandons l'affichage à l'écran de la chaîne de caractères "hello world!". Le C++ et sa bibliothèque standard, dont iostream fait partie, rend ceci très intuitif au moyen des objets std : :cout et std : :endl. [2] Cette utilisation des chevrons est caractéristique de la gestion de flux en C++ : on peut enchaîner les chevrons avec des objets de type différent, le C++ procèdera à une inférence de type pour vous.

On préfère "std : :endl" à "\n", car le caractère de retour chariot dépend de votre système d'exploitation. En effet "\n" ne fonctionne que sur UNIX, alors que le bon caractère est déterminé à la compilation avec son alternative de la bibliothèque standard du C++.

1. iostream = flux d'entrée/sortie

2. On appelle : : opérateur de résolution de portée. Dans l'utilisation ci-contre il permet d'aller chercher une fonction dans l'espace de nom std. On peut s'en passer en entrant l'instruction using namespace std ; après les directives du préprocesseur. Mais attention aux conflits de nom qui pourraient survenir avec d'autres bibliothèques !

2.2.3 Compilation du premier programme

Enregistrez votre travail, puis retournez à la console. Utilisez votre compilateur préféré, consultez son manuel au besoin. Pensez à l'autocomplétion !

```
clang helloworld.cpp -o helloworld.o
```

Cela réalise un exécutable binaire nommé "helloworld.o". NB : Atom permet de compiler au sein-même de son interface. Pour cela AFAIRE!!!!!!!

Vous pouvez à présent exécuter votre code :

```
./helloworld.o
```

2.3 Mécanismes du C++

2.3.1 Les types de données

2.3.2 Variables

A présent nous allons apprendre à gérer des variables en C++. Les variables ont un type : int, double, char ... Ce type doit être déclaré. Une variable possède une adresse en mémoire, à laquelle on peut accéder au moyen de l'esperluette &.

On peut affecter une valeur à une variable au moyen de l'opérateur "=". On peut effectuer des opérations sur les variables et les manipuler en tant qu'abstractions d'une quantité.

Voici un exemple de déclaration de variables, puis des exemples d'utilisations plus ou moins astucieuses de ces dernières. On y introduit des opérateurs utiles, vous comprendrez en lisant :

```
#include namespace std;
int main ()
{
  int a = 0;
  int b = 0;
  int c = 0;

  cout << "L'adresse mémoire de a, en hexadécimal, est: "
       << &a << endl;

  cout << "Les variables a,b et c contiennent respectivement "
       << a << " "
       << b << " "
       << c << " "
       << endl; // Nous avons poursuivi la même
                //instruction sur plusieurs lignes, sans ;

  a = a + 1 ; // Affecte la valeur a+1 à la variable a.
  b += 1 ;    // Ceci est une autre façon d'incrémenter b.
  c ++;       // Et encore une autre

  cout << "Maintenant, les variables a,b et c contiennent "
       << a << " "
       << b << " "
       << c << " "
       << endl;

  a /= 2; // Attention à la division entière !
  b = 8*c + 5;
  c = a + b;
  cout << "Les variables a,b et c contiennent "
       << a << " "
       << b << " "
       << c << " Pourquoi?"
       << endl;

  return 0;
```

```
}
```

La *portée d'une variable* est la taille de la plus grande portion de code dans laquelle elle est utilisable. Cette portée est limitée : à la fin de la section de code où elle est créée, une variable est automatiquement supprimée de la mémoire de l'ordinateur.

On profitera de ce mécanisme autant que possible : l'utilisation de variables globales peut rendre un code inextricable pour un nouvel utilisateur. La solution consistant à donner des noms à rallonge à vos variables globales est tout aussi délétère. Nous verrons comment travailler avec des variables locales à travers l'idéologie de programmation modulaire.

2.3.3 Types de données, Perte d'associativité

A FAIRE!!!!!!

2.3.4 Les boucles

Certaines tâches répétitives peuvent être effectuées au moyen de boucles. Les boucles for et while ont la syntaxe suivante :

```
for (int i = début; conditionDarret; queFaireEnFinDeBoucle)
{
  instruction1;
  instruction2;
  ...
}

int j=0;
while (conditionDeContinuationDeBouclePortantSurJ)
{
  instructions;
  penserAModifierJ;
}
```

Par exemple, si l'on veut calculer la factorielle d'un nombre :

```
int N = 10;
int factorielle = 1;
for (int n = 0; n < N; n++)
{
  factorielle *= (n+1);
}
cout << "La factorielle de "
     << N << " est "
     << factorielle;
```

Ou encore si l'on veut afficher à l'écran les entiers de 1 à 9 :

```
int i = 0;
while (i < 10)
{
  cout << i << endl;
  i++;
}
```

2.3.5 *Les fonctions*

Un programme gagne en lisibilité lorsqu'il est pensé de façon modulaire. En effet il utilise alors des variables dont la portée limitée facilite la compréhension du code petit bout par petit bout, tout en laissant la structure globale apparaître au travers des appels de fonctions.

Pour programmer de façon modulaire, on écrit plusieurs fonctions qui effectuent des tâches bien circonscrites, on combine alors ces fonctions dans la fonction main.

Il est important de rédiger un petit commentaire[3] descriptif au début de la fonction, de préférence en anglais, pour veiller à la relecture du code par vos successeurs. La programmation modulaire permet par ailleurs de distribuer les tâches entre programmeurs pour un travail d'équipe efficace.

3. On utilisera // pour ces commentaires. La syntaxe /* ... */ devrait être réservée à des fins de débugage.

La syntaxe générale d'une fonction est la suivante :

```
typeDeRetour nomDeFonction (type1 arg1, type2 arg2, etc )
{
  ...
  return objetDuTypeDeRetour;
}
```

Par exemple, si nous voulons calculer les carrés parfaits jusqu'à 10, nous pouvons écrire le code suivant :

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

double square(double x)
// Computes the square of x.
{
  return x*x;
}

int main ()
{
  cout << "Les 10 premiers carrés parfaits sont:";
```

```
  for (int i = 0; i<10; i++)
  {
    cout << square(i+1) << endl;
  }
  return 0;
}
```

On peut aussi différer l'implémentation de la fonction à condition d'écrire le *prototype de la fonction* avant son appel. Cela met en évidence la structure générale du code :

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

double square(double); // Le prototype de square déclare
                       // la fonction avant implémentation
int main ()
{
  cout << "Les 10 premiers carrés parfaits sont:";
  for (int i = 0; i<10; i++)
  {
    cout << square(i+1) << endl;
  }
  return 0;
}

double square(double x)
// Computes the square of x.
{
  return x*x;
}
```

Si vous consultez la documentation d'une bibliothèque C++ sur internet, ce sont parfois les prototypes des fonctions vous seront exposés. Ils seront accompagnés de plus ou moins de commentaires et exemples selon l'humeur des développeurs.

2.3.6 Passage par référence, passage par argument

On veut parfois qu'une fonction modifie les variables qui lui sont passées en argument. Cela est impossible avec le passage par argument précédent car une copie de chaque argument est réalisée en mémoire à l'appel de la fonction!

Pour cela, il faut effectuer un passage par référence. Le prototype de la fonction s'en trouve modifié :

```
void swap (int& a, int& b)
// échange les valeurs de a et b
{
  int c = a; // variable temporaire stockant la valeur de a
  a = b;
  b = c;
}
```

La commande "int &" désigne le type "référence vers un entier". Il contient l'adresse mémoire d'un entier. Une référence vers un objet se manipule comme l'objet lui-même. Pourtant, il y a une différence sémantique à écrire "a = 2" selon que a est un entier ou une référence vers un entier. Dans le premier cas, on veut dire "écris la valeur 2 dans l'adresse mémoire de la variable a", dans le second cas, on veut dire "écris 2 à cette adresse". Notez bien qu'il est impossible de réaffecter l'adresse d'une référence. Comment ferions-nous en effet : ce n'est pas ce que fait l'opérateur = comme nous venons de le voir![4]

4. Le fort typage des variables en C++ permet cette polysémie. Un même opérateur peut recevoir une implémentation différente en fonction du type des variables auxquelles il s'applique ; cela se nomme la *surcharge d'opérateurs* en C++.

2.3.7 *Pointeurs*

Une autre façon de travailler sur des adresses mémoires plutôt que des variables est d'utiliser des pointeurs. Ces objets contiennent des adresses mémoires, qui sont modifiables, au contraire des références. Ce concept est très utile dès lors que l'on veut faire de l'allocation dynamique de mémoire, c'est à dire allouer une quantité d'espace mémoire non connue à la compilation, mais connue à l'exécution. Par exemple quand on veut créer un tableau dont la taille est une variable.

Le prix à payer pour cette commodité est de devoir gérer soi-même l'allocation/libération de mémoire. L'opérateur new suivi d'un type d'objet demande au système d'exploitation de nous réserver une assez grande zone en mémoire pour écrire un objet du type demandé. Il renvoie l'adresse mémoire de celui-ci, que l'on peut stocker dans un pointeur.

L'opérateur delete agissant sur un pointeur informe le système d'exploitation qu'il peut à nouveau écrire à l'adresse que contient le pointeur. Il réinitialise ensuite le pointeur à l'adresse NULL, ne pointant sur aucune case mémoire.

Nous utilisons l'opérateur * pour manipuler les pointeurs , mais là encore, attention à la sémantique. Par exemple pour réaliser un tableau de doubles.

```
int main()
{
  int N = 10;

  double* T(0); //Pointeur sur un double, initialisé à 0
  T = new double[N]; // On alloue l'espace de N double
```

```
                    // On stocke l'adresse de début
                    // de cette zone mémoire dans T.

  *T = 3.14;  // Ecris la valeur 3.14 à l'adresse T
  cout << "L'adresse " << T << " contient " << *T << endl;

  for (int i=0; i<N; i++)
  {
    T[i] = 3.14*(i+1); // T[i] désigne l'adresse T, plus
                       // i fois la taille d'un double

    cout << "L'adresse " << T[i]
         << " contient " << T[i] << endl;
  }

  delete[] T; // Désalloue la mémoire qu'occupe T
              // Sans ça, on aurait une fuite de mémoire
              // A la fin de l'exécution.
  return 0;
}
```

Si vous oubliez les crochets de la fonction delete, vous ne libérerez que T[0] !

2.3.8 *Passer des arguments à la fonction main*

Bien des programmes admettent la saisie d'arguments à l'exécution. Cela permet de ne pas recompiler le code pour faire tourner le même programme en changeant un paramètre. Nous allons voir comment effectuer cette tâche dans un programme C++.

A l'exécution de "monProgramme arg1 arg2", l'interprète bash passe deux arguments à la fonction principale de monProgramme : un entier contenant le nombre d'arguments (ici 2), et un tableau statique [5] de pointeurs vers le type caractère [6]. On donne à ces deux arguments les noms de argc et arv, pour "argument count" et "argument values". Imaginons qu'un collègue à réalisé un programme qui prend en entrée un entier positif, et renvoie la suite des carrés parfaits jusqu'à cet entier, on ne mentionne que le prototype ici. On voudrait ne pas avoir à recompiler le programme :

5. Ce qu'indiquent des crochets

6. Nous avons vu qu'un pointeur de doubles permet de réaliser des tableaux de doubles. Il en est de même des pointeurs vers les caractères, que l'on peut voir comme des chaînes de caractères.

```
#include <iostream>   // Contient
#include <stdlib.h>   // Contient atoi
void afficheLesNPremiersEntiers(int);

int main (int argc, char* argv[])
{
```

```
  int n = 0;
  if (argc > 0)
  {
    n = atoi(argv[0]) // Conversion de char*
                      // vers un entier
    afficheLesNPremiersEntiers(n);
  }
  else
    cerr << "Vous devez entrer un nombre" << endl;
  return 0;
}
```

Après compilation de votre code, vous pouvez l'exécuter avec un passage d'argument ! Bien souvent, la partie pénible concernera la conversion du type char* vers le type que vous souhaitez. NB : le prototype de la fonction main nous oblige à utiliser des pointeurs vers des char pour la gestion des chaînes de caractères. Cela un reste du langage C, cependant le C++ implémente l'objet string, beaucoup plus ergonomique : ne vous privez pas de l'utiliser au besoin !

2.3.9 *Entrées/Sorties dans un fichier*

Vous serez probablement amenés à lire ou écrire dans des fichiers. Cette tâche étant très courante, elle est implémentée dans la bibliothèque standard du C++, dans la sous-bibliothèque fstream [7]. Une fois ouverte une instance d'un flux de fichiers, il suffit de s'y déplacer au moyen des opérateurs de gestion de flux.

7. File Stream, ou flux de fichiers.

Supposons que nous voulons écrire un fichier dont la première colonne représente une subdivision à 20 points de l'unité, et la deuxième colonne les valeurs de la fonction sinus évaluée entre ces points. Nous supposons qu'un collègue programmeur a réalisé deux fonctions : l'une reçoit en argument un pointeur alloué vers le type double et une taille de grille. Elle calcule la grille des abscisses entre $-\pi$ et π. L'autre reçoit deux pointeurs alloués vers le type double, leur taille, et évalue les sinus de la grille x sur la grille y. Ces deux fonctions renvoient un booléen indiquant comment s'est déroulée l'exécution.

```
#include <fstream>
using namespace std;

bool computeGrid(double* x, unsigned int N);
bool evaluateOnGrid(double* x, double* y, unsigned int N);

int main()
{
```

```
  int n = 20;
  double* x = new double[n];
  double* y = new double[n];

  if (!computeGrid(x,n))
     cerr << "Problème de calcul des abscisses" << endl;
  else
    if (!evaluateOnGrid(x,y,n))
       cerr << "Problème de calcul des ordonnées" << endl;

  ofstream monFichier ; // Déclaration d'un objet de type
                        // flux de fichier en écriture
  monFichier.open("../outputData/sinus.dat",ios::out);
                        // Ouverture de sinus.dat en écriture

  if (monFichier)
     {
       for (int i = 0; i<n; i++)
          monFichier << x[i] << " " << y[i] << endl;
       monfichier.close();
     }
  else
     cerr << "Erreur dans l'ouverture du fichier" << endl;

  delete[] x;
  delete[] y;
  return 0;
}
```

Remarques

1. Il faut bien réaliser que les fonctions réalisées par votre collègue ne modifient pas les pointeurs x et y mais écrivent simplement dans les adresses vers lesquelles ils pointent : point de passage par référence ici!

2. monFichier est une instance de la classe ofstream. Une classe, c'est un objet qui encapsule des fonctions auxquelles on peut accéder au moyen d'un ".". C'est le cas de open ici. De nombreuses bibliothèques proposent ce genre d'objets pratiques à l'utilisation.

3. On peut ensuite exécuter le programme, puis tracer le sinus à partir du fichier .dat au moyen du logiciel gnuplot. De retour dans bash, tapez :

```
cd ../outputData
gnuplot
plot "sinus.dat" with lines
exit
```

2.3.10 Compilation séparée : utilisation d'un makefile

BLABLABLA