Laboratoire 1

Bonjour Monsieur le Professeur

Ceci est le premier de deux laboratoires traitant principalement de divers aspects de base de la programmation en C. Il vous serait probablement profitable d'avoir bien lu le chapitre 1 des notes de cours avant de commencer.

1.1 Objectifs

1. Apprendre à vous brancher sur le réseau informatique de l'université;
2. Apprendre les bases de la gestion de fichiers et de répertoires sous le système d'exploitation LINUX;
3. Apprivoiser un programme d'édition de fichier, vous permettant d'écrire vos propres codes sources;
4. Apprendre à compiler et exécuter un code C simple, et détecter des erreurs de programmation;
5. Apprendre les bases structurelles d'un programme en C: canevas global d'un code, instructions de déclaration, instructions exécutables, types de variables, commentaires, etc.;
6. Apprendre les bases de l'interaction usager-programme (instructions C `scanf`, `printf`, etc.).

1.2 Rapport de laboratoire

Il n'y a pas de rapport à remettre pour ce labo! Mais vous ne sortirez pas d'ici avant qu'on ait vu vos trois codes C exécuter correctement...

1.3 Se brancher sur ESIBAC

ESIBAC est un des systèmes informatiques "public" à l'UdeM, sur lequel vous devrez vous brancher pour faire tous les labs dans ce cours. On vous expliquera en détail comment faire au début de cette session de laboratoire. En bref, un compte est déjà crée pour chacun(e) d'entre vous, et restera actif jusqu'à la fin décembre. Pour plus d'information sur ESIBAC, voir:

`http://www.dgtic.umontreal.ca/esi/esibac.html`

`http://www.bac-esi.umontreal.ca`

Ce qui suit présuppose que vous êtes déjà en mesure de vous brancher sur votre profil via le portail UdeM, et que vous avez défini un mot de passe général. Si ce n'est pas fait, demandez à un des démonstrateurs on vous montrera comment.

À partir des ordis du M-635, malheureusement montés en Windows, la communication sur ESIBAC se fait via une fenêtre dite "terminal". Une fois votre session Windows lancée, cliquez sur le dossier *logiciels avancés*, puis sur *hummingbird*, puis sur *ssh_esilbac1.xs*. Ceci devrait faire apparaitre une fenêtre terminal à l'écran. De cette fenêtre vous avez accès aux compilateur C.

Les fichiers que vous créerez sur votre compte ESIBAC peuvent être accédés de n'importe quel autre ordi ou terminal branché au réseau universitaire public, y compris à la salle d'ordi réservé aux étudiant(e)s du département de physique, soit le G-609 au Pavillon Roger-Gaudry (code d'entrée 3-4-5).

1.4 Fichiers et répertoires

À moins de n'avoir jamais touché à un ordi de votre vie, vous êtes probablement familier(ère)s avec l'idée d'un **fichier** (ou "document"), et de leur organisation en **répertoires** (ou "dossiers", dans l'univers Microsoft). Une organisation intelligente de vos programmes sources, fichiers de données, fichiers de sortie, etc., est un saine habitude de travail, et vous aidera grandement à vous retrouver dans vos affaires rendu à la onzième semaine de labo...

La Figure 1.1 illustre ce qui pourrait bien être le contenu du compte de l'usager `milou` à mi-chemin durant le second laboratoire. Sous LINUX le répertoire-mère de l'usager `milou` s'appelera toujours `/home/milou`. Ce répertoire content ici trois répertoires, dont un (`labo2`) qui en contient lui-même un autre, ainsi que divers fichiers, et un autre demeure présentement vide (`labo3`). Le répertoire-mère contient également trois fichiers quelconques en plus des trois répertoires. La gestion de vos fichier exige que vous puissiez créer des répertoires, y déplacer des fichiers, en éliminer les fichiers désuets, etc. Vous avez peut-être déjà effectué ce genre d'opérations via la souris, directement ou à l'aide de menus. Dans l'environnement de type "terminal" fourni par ESIBAC, vous aurez à utiliser les commandes de bases du système d'exploitation LINUX. Le Tableau 1.1 en liste les plus importantes; commencez par mémoriser celles-là, et vous apprendrez graduellement les autres "sur le tas".

Table 1.1: Commandes de base en Linux

Commande/Usage	Fonction	Notez Bien
`ls` *rep*	liste fichiers dans répertoire *rep*	si *rep* omis, liste répertoire courant
`cd` *rep*	change au répertoire *rep*	si *rep* omis, ramène à `/home/usager`
`cp` *nom*1 *nom*2	crée copie *nom*2 du fichier *nom*1	fichier *nom*1 demeure
`mv` *nom*1 *nom*2	assigne le nom *nom*2 au fichier *nom*1	*nom*1 disparait
`cp` *nom*1 *rep*	copie fichier *nom*1 dans *rep*	*rep* doit être dans rép. courant
`mv` *nom*1 *rep*	déplace *nom*1 dans répertoire *rep*	*rep* doit être dans rép. courant
`mv` *rep*1 *rep*2	déplace répertoire *rep*1 dans *rep*2	*rep*2 doit être dans rép. courant
`rm` *nom*1	efface le fichier *nom*1	
`more` *nom*1	liste fichier *nom*1 à l'écran	
`mkdir` *rep*	crée répertoire *rep* dans rép. courant	
`rmdir` *rep*	efface le répertoire *rep*	*rep* doit être vide
`pwd`	liste le nom du répertoire courant	

Notez bien que plusieurs de ces commandes sont définies de manière relative au répertoire courant. Ainsi, une fois branché l'usager `milou` a deux options pour examiner le contenu de son répertoire `labo1`:

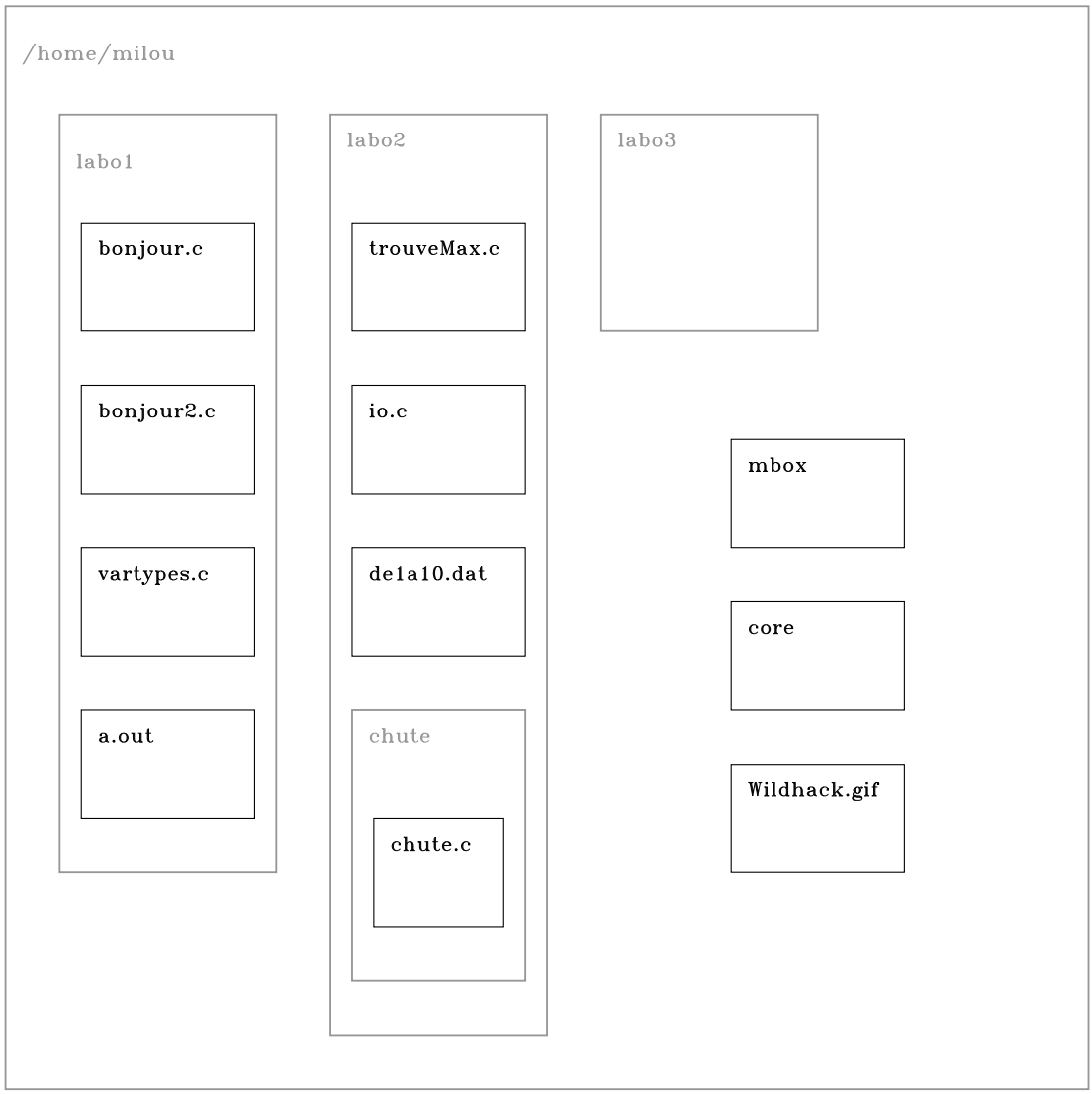

Figure 1.1: Structure typique des fichiers et répertoires de l'usager `milou`, au moment du second laboratoire. Les encadrés gris représentent des répertoires, et ceux en noir des fichiers. Au moment où vous vous branchez sur votre compte, vous atterissez immédiatement et automatiquement dans un répertoire appellé `/home/usager`, où "`usager`" est votre propre identificateur personnel (`milou` dans le cas présent; pour vous, ce sera un code alphanumérique commençant par `p0...`). Ici ce "home" contient trois répertoires, eux-même contenant des fichiers et/ou d'autres répertoires (comme `labo2`, qui ici contient un répertoire appelé `chute`, qui lui-même contient un fichier appelé `chute.c`), ainsi que trois fichiers individuels. Une série de commande Linux, décrites au Tableau 1.1, vous permet de gérer fichiers et répertoires (création/destructions de répertoires, copie et déplacement de fichiers, etc).

1. `cd labo1` *return* `ls` *return*

2. `ls labo1` *return*

ATTENTION: ici et dans tout ce qui suit, "*return*" veut dire "presser la touche marquée "Return" (ou "Enter") sur votre clavier; ça ne veut pas dire de taper la suite de caractères `r e t u r n` !! La commande `cd` permet de "descendre" dans la structure des répertoires; pour "remonter"; par exemple, si le répertoire courant est `chute` et qu'on désire se déplacer dans le répertoire `labo1`, on a encore plusieurs options équivalentes:

1. `cd /home/milou/labo1` *return*

2. `cd` *return* `cd labo1` *return*

3. `cd ../../labo1` *return*

Ici le "`../../labo1`" indique qu'on doit remonter de deux niveaux à partir du répertoire courant, soit `chute`→`labo2`→`/home/milou`, puis descendre à `labo1`. Notez ici que le répertoire-cible `labo1` peut être identifié en terme absolu à partir du répertoire `/home/milou` (option 1), ou de manière relative par rapport au répertoire courant (option 3). Le mêmes règles s'appliquent aux autres commandes comme `cp` ou `mv`; ainsi, si on est dans le répertoire `/home/milou`, toutes les instructions suivantes auront comme effet de faire une copie dans le répertoire `labo2` du fichier `bonjour.c` contenu dans le répertoire `labo1`:

1. `cp labo1/bonjour.c labo2/bonjour.c` *return*

2. `cd labo1` *return* `cp ../labo2/bonjour.c` *return*

3. `cd labo2` *return* `cp ../labo1/bonjour.c .` *return*

4. `cp /home/milou/labo1/bonjour.c /home/milou/labo2/.` *return*

Notez dans la troisième option que le second argument de la commande `cp` est un simple point, qui représente un raccourci voulant dire "ici et sous le même nom". La quatrième option illustre comment, en utilisant une identification absolue des fichiers, on peut copier ou déplacer n'importe quoi n'importe où, et à partir de n'importe quel répertoire courant.

Votre première tâche sera donc de vous créer un répertoire sous votre répertoire-mère `/home/usager`, dans lequel vous regrouperez et conserverez tous les fichiers contenant les petits codes C que vous aurez à écrire dans la suite de ce labo.

1.5 Un premier code C

Il s'agit maintenant de faire tourner notre premier code en C. C'est une procédure en trois étapes: (1) écriture du code source en C, (2) compilation en langage-machine, et (3) exécution du programme.

Démarrez le programme d'édition de fichier, en tapant dans la fenêtre ESIBAC la commande:

`kate` & *return*

L'ajout du caractère "&" assure ici l'exécution du processus en "arrière-plan", ce qui permet de conserver le contrôle de la fenêtre "commande" dans laquelle la commande "kate" a été invoquée. Dans la fenêtre d'édition qui apparaitra alors, tapez, ligne par ligne, le petit code C suivant:

```
#include <stdio.h>
int main(void)
{
  /* Ce programme dit "bonjour" a l'ecran */
  printf ("Bonjour Monsieur le Professeur\n") ;
}
```

La ligne `#include <stdio.h>` indique que le programme qui suit utilisera une ou plusieurs fonctions ou commandes prédéfinies contenue dans la librairie `stdio.h`, une des nombreuses librairies standards du langage C. Certains compilateurs tolèrent l'omission de cette ligne, mais ce sera une bonne habitude de toujours l'inclure, afin d'éviter des surprises si vous travaillez sur différents ordis avec les même codes sources. L'instruction `int main(void)` identifie ce qui suit entre les parenthèses curvilignes {...} comme étant le **programme principal**, par opposition à une **fonction**, concept avec lequel nous ferons connaissance la semaine prochaine (et qui fournira l'occasion de clarifier ce que veulent dire les `int` et `(void)`...). Ici le programme ne contient que deux lignes d'instruction, la première étant une ligne de commentaires, délimitée par les `/* ... */`, qui ne fera absolument rien au moment de l'exécution, et une ligne d'instruction `printf`, dont l'exécution écrira à l'écran tout ce qui est contenu entre les `" ... "`, sauf le `\n` qui produit un saut de ligne final.

Maintenant sauvegardez le fichier en cliquant sur l'icône approprié au haut de la fenêtre d'édition. Pour un nouveau fichier, le programme d'édition vous demandera de spécifier un nom à assigner au fichier. Le code source doit porter un nom qui l'identifiera de manière unique parmi les autres fichiers pouvant être contenus dans le répertoire de travail. Traditionnellement un code source en langage C se terminera par le suffixe "`.c`". On pourrait par exemple appeler `bonjour.c` le fichier contenant le code C ci-dessus.

Pour compiler ce code, revenez à la fenêtre ESIBAC et dans le même répertoire contenant le fichier source, tapez:

`gcc bonjour.c` *return*

Ceci créera dans le répertoire courant un **fichier exécutable** appelé par défaut `a.out`, qui est par la suite exécuté en tapant simplement

`./a.out` *return*

Ceci devrait produire à l'écran la sortie suivante:

```
Bonjour Monsieur le Professeur
```

Celà n'a pas l'air de grand chose, mais ce que vous avez accompli à date est déjà très substantiel: vous brancher, écrire un code source, le sauvegarder sur disque, le compiler, et l'exécuter. Bravo!

1.6 Un second code C

On continue. Créez maintenant un second code source qui a l'air de ceci:

```
#include <stdio.h>
int main(void)
{
  /* Ce programme vous dit "bonjour" a l'ecran */

  /* Declarations ------------------------------ */
  char votreNom[30] ;

  /* Executables ------------------------------- */

  printf ("SVP tapez votre nom : ") ;
```

```
  scanf  ("%s", &votreNom) ;
  printf ("Bonjour %s\n", votreNom) ;
}
```

Le code compte maintenant trois instructions exécutables, qui seront exécutées l'une après l'autre, selon l'ordre dans lequel elles apparaissent dans le code source. L'instruction `printf` vous connaissez déjà; l'instruction `scanf` fait l'inverse, elle lit quelquechose que l'usager tape à l'écran au moment de l'exécution. Notez que chaque ligne d'instruction se termine par un point-virgule "`;`", qui indique explicitement la fin d'une instruction. Ceci permet de combiner plusieurs instructions courtes sur la même ligne, ou d'étaler une très longue instruction sur plusieurs lignes. Ceci ne change absolument rien à l'exécution du code, mais peut en améliorer la lisibilité.

Toujours dans le département des nouveautés, notez qu'ici on définit une variable appelée `votreNom`, de type caractère et pouvant contenir ici un maximum de 30 caractères (instruction commençant par `char`). Il est obligatoire en C de regrouper toute les définitions de variable au début du code, avant les instructions exécutables. Cette variable aurait pu avoir n'importe quel nom (`maman`, `bozo`, `proutschniaque`,...) sans affecter l'exécution, tant que le même nom de variable apparait aux instructions de déclaration et d'exécution. Il est cependant d'usage de donner aux variables des noms reliés à leur sens dans la logique du programme. Notez également qu'en C le nom d'une variable:

1. peut être composé des 26 caractères de base de l'alphabet, des chiffres de 0 à 9, ou du caractère "`_`"; cependant, pas de caractères accentués du clavier français!

2. ne doit pas cependant commencer par un chiffre; ainsi, `2freresDupondt` n'est pas un nom de variable légal en C;

3. ne peut faire plus de 32 caractères; en utiliser plus de 32 ne cause pas d'erreur de compilation, mais les surnuméraires sont perdues.

4. ne peut contenir d'espaces blancs;

5. distingue les majuscule des minuscules; ainsi, les variables `votreNom`, `VotreNom`, et `votrenom` sont considérées par le compilateur comme trois entités distinctes, devant chacune être déclarée, initialisée, etc.

Il faut finalement noter que le C réserve certaines chaines de caractères pour définir des **mots-clefs** qui jouent un rôle spécifique au niveau de la programmation; si il vous vient la brillante idée de définir une variable ou une fonction ayant le même nom, il pourrait vous arriver des choses É-POU-VAN-TA-BLES !! Voici une liste alphabétique de ces mots-clefs à éviter:

```
auto, break, case, char, const, continue, default, do, double, else, enum,
extern, float, for, goto, if, int, long, register, return, short, signed,
sizeof, static, struct, switch, typedef, union, unsigned, void, volatile,
while.
```

Revenant au petit code ci-dessus, examinez bien la seconde instruction `printf`, en la comparant attentivement avec la première. Le "`%s`" apparaissant entre les guillemets indique que la variable à imprimer, ici `votreNom`, est une chaine de caractères ("`s`" pour "string"). Nous verrons sous peu qu'il existe d'autres descripteurs de format pour d'autres types de variables.

Finalement, vous noterez que deux lignes vides ont été insérées pour démarquer les parties déclaration et exécution du code. Ces lignes ne sont pas vues à la compilation, en ne font qu'améliorer la lisibilité du code pour l'usager.

Compilez et exécutez ce programme. Une fois convaincu(e) que tout fonctionne comme prévu, expérimentez un peu:

1. Enlevez un des point-virgules quelquepart; essayez de compiler et exécuter.
2. Introduisez une faute de frappe dans le nom de la variable `votreNom` dans la section déclaration; essayez de compiler et exécuter.
3. Remplacez le "`%s`" par "`%f`" dans la seconde instruction `printf`; essayez de compiler et exécuter.
4. Enlevez le "`[30]`" dans la déclaration de la variable `votreNom`; essayez de compiler et exécuter.
5. Déplacez la seconde instruction `printf` au début de la section exécutable (i.e., *avant* l'instruction `scanf`); essayez de compiler et exécuter.

Ces divers exemples devraient vous faire réaliser qu'il y a trois types distincts "d'erreurs" qui peuvent se produire:

1. **Erreur de compilation**: votre code ne respecte pas les règles syntaxiques du langage C; le compilateur stoppe sans produire de fichier exécutable;
2. **Erreur d'exécution**: la syntaxe de votre code est valide; mais quelquechose dans la logique du code conduit à une erreur au moment de l'exécution.
3. **Erreur conceptuelle:** le programme compile et exécute sans produire de message d'erreur, mais le résultat est fautif.

Avec un peu d'expérience, le premier type d'erreur devient habituellement facile à retracer à partir des messages d'erreur produits au moment de la compilation. Les erreurs d'exécution peuvent l'être pas mal moins, et les erreurs de type conceptuel encore moins. Notez qu'une erreur durant l'exécution produira habituellement un fichier appelé `core` dans le répertoire où se trouve l'exécutable `a.out`. Ce fichier contient une image de la mémoire du processeur au moment de l'arrêt du programme, et peut être utilisé en entrée à un programme dit de "déverminage" (ou "debugger" en bon anglais). Nous ne ferons pas usage de dévermineur dans le cadre de ce cours, mais ce serait une bonne habitude de toujours éliminer les fichiers `core` suite à une erreur d'exécution, car ils peuvent parfois avoir une taille susbtantielle.

1.7 Un troisième code C

Le dernier petit code C avec lequel vous travaillerez dans le cadre de ce premier labo vise à vous faire explorer les comportements parfois particuliers de divers opérateurs arithmétiques agissant sur divers types de variables. Relancez votre éditeur, et tapez le code suivant:

```
#include <stdio.h>
int main(void)
{
  /* Ce programme joue avec les types de variables */

  /* Declarations ---------------------------------- */
   int   i, j, k ;
   double x, y, z ;

  /* Executables ----------------------------------- */
  i=3 ; j=7 ; k=5 ;
  x=3.; y=7.; z=5.;

  printf ("Multiplication  i*j = %d\n", i*j) ;
```

```
  printf ("Division        i/j = %d\n", i/j) ;
  printf ("Division        j/i = %d\n", j/i) ;
  printf ("Multiplication  x*y = %f\n", x*y) ;
  printf ("Multiplication  x*y = %e\n", x*y) ;
  printf ("Combinaison   x+y/z = %f\n", x+y/z) ;
  printf ("Combinaison (x+y)/z = %f\n", (x+y)/z) ;
  printf ("Combinaison   x/y*z = %f\n", x/y*z) ;
  printf ("Combinaison x/(y*z) = %f\n", x/(y*z)) ;
  printf ("i Modulus j         = %d\n", i%j) ;
  printf ("j Modulus i         = %d\n", j%i) ;
  printf ("j Modulus i         = %f\n", j%i) ;
  printf ("Division        x/y = %d\n", x/y) ;
}
```

Les lignes d'instruction dans la section déclarations qui commencent par `int` et `float` identifient les variables dont les nom suivent comme étant des entiers et des réels, respectivement. Une fois compilé et exécuté, ceci devrait produire la sortie suivante:

```
Multiplication  i*j = 21
Division        i/j = 0
Division        j/i = 2
Multiplication  x*y = 21.000000
Multiplication  x*y = 2.100000e+01
Combinaison   x+y/z = 4.400000
Combinaison (x+y)/z = 2.000000
Combinaison   x/y*z = 2.142857
Combinaison x/(y*z) = 0.085714
i Modulus j         = 3
j Modulus i         = 1
j Modulus i         = 0.000000
Division        x/y = 1071345078
```

(où l'entier résultant de la dernière division pourrait bien avoir une autre valeur que celle reproduite ici, dépendant de l'ordi et compilateur utilisés). Il y a plusieurs choses à remarquer et comprendre ici:

1. La multiplication est effectuée via le symbole "`*`", et la division via "`/`"; ces opérateurs peuvent agir soit sur des entiers, soit sur des réels.

2. Les opérations arithmétiques s'effectuent de gauche à droite, mais la multiplication et la division ont priorité sur l'addition et la soustraction; comparez et comprenez bien les résultats du calcul de `x+y/z` versus `(x+y)/z`, et de `x/y*z` versus `x/(y*z)`, et comment l'usage judicieux des parenthèses permet de contourner ces conventions.

3. La division de deux variables déclarées comme des entiers produit un résultat étant lui-même un entier, donc tout reste de la division est perdu (ici, $3/7 = 0$ et $7/3 = 2$).

4. Le reste de la division de deux entiers peut être calculé explicitement avec l'opérateur "`%`", dit "modulo". Assurez vous de bien comprendre ici pourquoi $3\%7 = 3$ et $7\%3 = 1$.

5. À chaque type de variable doit correspondre le bon **descripteur de format**, sinon la sortie peut faire n'importe quoi (regardez bien et comparez les deux dernière ligne de la sortie avec les instructions `printf` les ayant produite).

6. Deux formats de sortie sont disponibles pour les variables réelles, soit la notation décimale habituelle (produite par le descripteur `%f`) et la notation exponentielle (produite via `%e`):

$$\texttt{2.100000e+01} \quad \equiv \quad 2.1 \times 10^1 \quad \equiv \quad \texttt{21.00000}$$

Le Tableau ci-dessous liste les différentes déclarations légales de variables en C. Notez qu'il existe trois types d'entiers et de réels, dépendant du niveau de précision requis dans la représentation numérique (voir chapitre 1 dans les notes de cours). Dans la majorité des cas, l'usage de `int` pour les entiers et `float` pour les réel devrait suffire (voir cependant le chapitre 2 des notes de cours!), et les types `short` et `long double` ne sont que très rarement utilisés.

Table 1.2: Types de variables

Déclaration	type	descripteur
`short`	entier "court"	`%d`
`int`	entier	`%d`
`long`	entier "long"	`%d`
`float`	réel	`%f`, `%e`
`double`	réel double précision	`%f`, `%e`
`long double`	réel quadruple précision	`%f`, `%e`
`char`	caractère	`%c`, `%s`

Vous aurez peut-être remarqué que la commande `printf` agissant sur des variables de type `float` via les descripteurs de format `%e` et `%f`, ne produit que sept chiffres significatifs en sortie; pour des raisons cosmétiques (ou autres), il est souvent utile de pouvoir contrôler le nombre de chiffres significatifs en sortie. Ceci peut se faire en ajoutant un préfixe numérique aux descripteurs. Par exemple, les instructions suivantes:

```
printf ("Combinaison x/(y*z) = %10.3f\n",x/(y*z)) ;
printf ("Combinaison x/(y*z) = %10.3e\n",x/(y*z)) ;
printf ("Combinaison x/(y*z) = %20.10e\n",x/(y*z)) ;
printf ("Combinaison x/(y*z) = %12.10e\n",x/(y*z)) ;
```

vont produire en sortie:

```
Combinaison x/(y*z) =      0.086
Combinaison x/(y*z) =  8.571e-02
Combinaison x/(y*z) =     8.5714288056e-02
Combinaison x/(y*z) = 8.5714288056e-02
```

La convention des préfixes est bizarroide à souhait (et héritée du FORTRAN); `%10.3e` indique que la sortie est en notation exponentielle compte dix chiffres ou caractères dont 3 sont réservés aux décimales suivant le point; traditionnellement le 10 doit inclure ici le "`.`" et le ("`e+`", donc un descripteur `5.3e` ne produirait pas une sortie valide; de même, essayer d'imprimer le chiffre 100.2 avec le descripteur `5.3f` produirait une sortie incorrecte; on a besoin ici d'au moins 5 caractère/chiffres, puisque le "`.`" compte encore. Et dans les deux cas, si le `float` à imprimer est négatif, le "`-`" doit également être comptabilisé dans le total. La majorité des compilateurs C peuvent contourner ce problème, mais les sorties se retrouvent décalées horizontalement sur la page, comme dans l'exemple ci-dessus; ce genre d'absurdités ne devrait pas trop vous causer de problèmes, à moins d'être quelquepeu maniaque et insister sur le fait que les sorties à l'écran soient toutes joliment alignées, etc.

Vous pouvez maintenant modifier le code ci-dessus et expérimenter avec les divers types de variables, leur impact sur les opérations arithmétiques, l'utilisation des divers descripteurs de format, etc. Faites-vous la main là-dessus une dizaine de minutes, demandez à un des démonstrateur de vous tester, et ensuite voilà, le premier labo est terminé. Caramba!

Lectures supplémentaires:

Notes de cours: Chapitre 1

Delannoy — premier langage : Chapitres 1, 2 et 3

Delannoy — programmer en C : Chapitres 1 et 2