Initiation à la programmation pour les sciences.

juin 2000 environ

Table des matières

*Ce manuscrit contient des contributions de Renée de Graeve, Frédéric Faure, Jonathan Ferreira, Bernard Parisse, Philippe Peyla Gérard Vinel,

1 Généralités.

1.1 Mémoire, formats de données.

Lorsqu'on utilise un ordinateur, on utilise en fait des programmes qui manipulent des données. Le programme est exécuté par le microprocesseur (en anglais *CPU* i.e. Central Processing Unit) de l'ordinateur. Pour que le microprocesseur puisse accéder aux données à manipuler par un programme, ces données doivent être placées dans la mémoire de l'ordinateur. La mémoire est constitué de transistors toutes identiques qui peuvent avoir deux valeurs 0 ou 1. On les regroupe souvent par 8 en formant un *octet* (en anglais *byte*), qui peut prendre $2^8 = 256$ valeurs. Chaque octet possède un numéro ce qui permet au microprocesseur de les distinguer, ce numéro s'appelle l'adresse mémoire. Par exemple, notre serveur étudiant dispose d'un giga-octet (1 GO), c'est-à-dire 2^{30} soit environ un milliard d'octets. Cette mémoire est appelée RAM : random access memory, ce qui signifie que l'on peut accéder à n'importe quelle cellule mémoire (contrairement à une bande magnétique par exemple où il faut d'abord se déplacer au bon endroit).

Lorsque l'exécution d'un programme est terminée, les données manipulées par ce programme n'ont plus besoin d'être en mémoire, on les stocke alors dans un disque dur, sur un CD-ROM ou sur une disquette et on libère la place pour d'autres programmes et données.

La mémoire permet donc de stocker facilement des entiers compris entre 0 et 255. Mais on souhaite pouvoir manipuler des données d'un type différent, par exemple des lettres (pour pouvoir écrire un e-mail par exemple), des images, des listes, des tableaux, ... Il est alors nécessaire d'établir une correspondance entre ce type de données et un ou plusieurs entiers compris entre 0 et 255. Par exemple on pourrait décider de représenter la lettre A par l'entier 0, la lettre B par l'entier 1, ..., la lettre Z par l'entier 26. Dans la réalité les ordinateurs utilisent plutôt le codage ASCII qui représente A par 65, B par 66, etc.. Pour représenter un texte on utilisera alors une suite consécutive d'entiers de la mémoire. De plus, on peut souhaiter enrichir la mise en page en utilisant des tailles de caractères différents ou des caractères en gras ou en italique ce qui complique la représentation des données. Pour une image, on peut représenter chaque pixel par une suite de 3 entiers représentant l'intensité de la couleur rouge, bleue, verte de ce pixel. A chaque type de donnée correspond donc un format de numérisation permettant de stocker cette donnée en mémoire à l'aide d'un ou plusieurs entiers compris entre 0 et 255. Pour un programmeur il est essentiel de toujours savoir quel est le type des données que l'on manipule à un instant donné (une lettre, une suite de lettres, un entier, un nombre réel, etc.)

1.2 Programme, langage et compilateur.

Un programme est constitué par une suite d'instructions chacune très simples (par exemple additionner deux entiers compris entre 0 et 255). Les instructions sont exécutées les unes à la suite des autres, certaines instructions permettent de rompre ce flot d'exécution (branchements, boucles, appel de sous-programmes). La durée d'une instruction se mesure en nombre de cycles de l'horloge interne du microprocesseur, la vitesse de cette horloge se mesure en hertz, par exemple un processeur cadence a 1GHz exécute 1 milliard de cycles d'horloge par seconde.

Les instructions du microprocesseur sont codées par un ou plusieurs nombres entiers afin de pouvoir stocker en mémoire un programme pour pouvoir l'exécuter. Ce codage est le jeu d'instructions du microprocesseur. Chaque microprocesseur possède son propre jeu d'instructions. Pour l'ordinateur, un programme est une suite d'instructions donc une suite d'entiers codant ces instructions appelé *code compilé*. Un programme est donc une donnée, mais d'un type très particulier (appelé exécutable) puisque les données doivent correspondre à des instructions du microprocesseur dans un ordre qui correspond à un but précis, l'exécution du programme.[1]

Pour le concepteur d'un programme, écrire les nombres correspondant aux instructions du microprocesseur présente de nombreux désavantages :

- une action simple (par exemple lire un caractère frappé au clavier ou afficher un caractère à l'écran) nécessite plusieurs dizaines voir plusieurs centaines ou millers d'instructions du microprocesseur, écrire un programme simple de cette manière est fastidieux,
- pour un même programme, il faut réécrire une suite d'entiers différente pour chaque processeur
- écrire une suite de nombres comme 1, 255,34, 35, 240, 17, 125, ... n'a pas de sens direct pour un humain alors qu'écrire "affiche le caratère A" a un sens, le risque d'erreur est donc très important

On utilise donc un langage intermédiaire pour décrire les instructions d'un programme appelé langage de programmation. Ce langage est facile à lire et à modifier par l'être humain. Le programmeur écrit une suite d'instructions dans ce langage, on appelle cette suite d'instructions le *source* du programme.

Le source d'un programme se présente comme un fichier texte sans effets de mise en page, caractères gras ou autres. Pour le créer on utilise un logiciel appelé *éditeur de texte* (à ne pas confondre avec un logiciel de *traitement de texte*, il ne faut jamais utiliser un logiciel de traitement de texte pour écrire un fichier source). Le texte source contiendra par exemple la phrase "affiche le caractère A" traduite dans le langage, en C++, cette instruction se traduirait par :

```
cout << 'A' ;)
```

Puis un autre logiciel, le *compilateur*, va traduire le langage de programmation en une suite d'instructions directement compréhensible par l'ordinateur.[2]

[1] On peut ici faire un parallèle avec le code génétique. Le role du code exécutable est joué par le code ADN, le rôle du microprocesseur est joué par le ribosome qui synthétise une protéine à partir du code ADN.

[2] Si on continue le parallèle avec le code génétique, on peut dire que décoder le génome correspond à inventer un langage permettant de comprendre facilement le rôle biologique de tel ou tel morceau de chromosome, ce langage "compilé" étant le code ADN.

1.3 Système d'exploitation et système de fichiers.

On a vu dans les sections précédentes que la création d'un programme nécessite plusieurs étapes :

- écrire le texte source du programme
- le compiler avec un compilateur
- exécuter le programme

La réalisation de chacune de ces étapes nécessite de pouvoir exécuter un logiciel dédié à cet usage et de pouvoir manipuler les données correspondantes sur le disque dur ou sur une disquette (le texte source du programme et le programme compilé). C'est le rôle du système d'exploitation d'un ordinateur et des systèmes de fichiers pour les données qu'il stocke.

1.3.1 Système de fichiers.

Avant d'être stockées par exemple sur le disque dur, les données sont regroupées en un ensemble appelé fichier (en anglais *file*). Chaque fichier porte un nom. Les fichiers sont eux-mêmes organisés en ensembles appelés répertoires (en anglais *directory*) selon une structure arborescente. Chaque noeud de l'arborescence est un répertoire dans lequel on trouve des fichiers ou d'autres répertoires. Dans les systèmes de fichiers Unix, les noms des répertoires "noeuds" de l'arborescence sont séparés par le signe `/`. Par exemple `/h/moore/u4/phyliphrec/aaal/toto.cc` signifie que le fichier `toto.cc` se trouve dans le répertoire `aaal` qui se trouve lui-même dans le répertoire `phyliphrec`, qui se trouve lui même dans ...

Les noms de fichiers sont souvent composés d'un premier mot suivi d'un point et d'une "extension" de 1 à 3 lettres qui donne une indication sur le format de numérisation ou l'usage du fichier. Par exemple le nom `toto.cc` est composé du mot `toto` et de l'extension `cc` qui signifie que le fichier est un programme écrit en C++.

1.3.2 Systèmes d'exploitation.

Il s'agit du logiciel qui permet en particulier d'accéder au système de fichiers et de lancer des programmes. L'interaction entre l'être humain et le système d'exploitation peut se faire de plusieurs manières : par l'intermédiaire d'une interface graphique[3] ou par l'intermédiaire de commandes interprétées par un logiciel appelé *interpréteur de commandes* ou *shell*. Les interfaces graphiques sont souvent plus simples à manipuler par un utilisateur débutant, mais elles présentent un inconvénient majeur : l'effet correspondant à une action donnée (par exemple un click avec le bouton du milieu de la souris) dépend de l'interface graphique car l'action en elle-même n'a pas de sens intrinsèque, l'effet d'une telle action évolue parfois avec les versions successives d'une même interface graphique. Les interpréteurs de commandes sont plus difficiles à manipuler au premier abord, mais les effets correspondent à l'utilisation de *noms* des commandes, ils ne changent pas au cours du temps et sont utilisables sur des ordinateurs différents (par exemple les commandes Unix fonctionnent sous GNU/Linux, Mac

[3] On parlera ici d'interface graphique lorsque l'interaction entre le logiciel et l'ordinateur ne se limite pas à échanger des données alphanumériques. Par exemple, lorsqu'on utilise une souris.

OS X, Solaris, HP-UX, FreeBSD, etc... mais également sous Windows avec les outils `cygwin` qu'on peut récupérer gratuitement sur `http://sources.redhat.com/cygwin`).

1.4 Échange de données et réseau.

On a vu que chaque logiciel doit numériser les données qu'il manipule pour les stocker en mémoire, en utilisant un *format de données*. Pour que les logiciels puissent communiquer entre eux, ils doivent utiliser des formats de données communs. Par exemple les caractères alphanumériques sont souvent numérisés selon le format standard ASCII selon la règle suivante : 0 correspond au code 48, 1 à 49, etc., A correspond au code 65, B à 66, etc..

L'échange de données entre des logiciels situés sur des ordinateurs différents se fait en établissant une connexion entre les ordinateurs, on crée ainsi un réseau d'ordinateurs, par exemple le réseau Internet. Sur le réseau Internet, chaque ordinateur reçoit un nom composé de deux parties séparées par un point (`.`) : le nom de la machine (par exemple `pcm2`) et le nom de domaine (par exemple `e.ujf-grenoble.fr`). Sur Internet, les échanges entre ordinateurs se font selon des règles appelées protocoles qui correspondent à l'usage que l'on fait des données (par exemple `HTTP` pour le web, `SMTP` pour la messagerie, `FTP` pour l'échange de fichiers, etc.). Pour échanger des fichiers entre deux ordinateurs, il faut donc utiliser un même protocole pour communiquer physiquement les données et utiliser des formats de données compatibles pour que les logiciels sachent interpréter les données transmises (cela parait évident, mais pensez-y lorsque vous envoyez par courrier électronique un fichier attaché : par exemple si vous envoyez un fichier au format Word, il est possible que le destinataire ne possède pas de logiciel permettant de lire ce format).

1.5 Aspects légaux.

Les logiciels et les données numérisées ne sont pas des biens matériels : on n'achète jamais un logiciel, on achète un droit d'utilisation appelé licence, c'est un contrat entre le détenteur des droits d'auteur et l'utilisateur final. La copie de données ou de logiciels est donc interdite sans autorisation explicite de l'auteur (sous peine d'amende d'un montant pouvant atteindre 300 000 euros ou/et de peines d'emprisonnement). Certains auteurs de logiciels et/ou documents donnent le droit d'utiliser et modifier leurs oeuvres en utilisant des licences prévues à cet effet, comme la FDL (Free Documentation Licence) utilisée pour ce document, ou la GPL (General Public License : cf. `www.gnu.org`) utilisée par de nombreux logiciels. Les logiciels placés sous des licences de ce type sont appelés logiciels libres et peuvent être légalement copiés. Dans ce module on utilisera chaque fois que cela est possible des logiciels libres, c'est le cas en particulier du système d'exploitation de notre serveur (GNU/Linux), du compilateur (g++), etc.

1.6 Exercices.

Donner des exemples de logiciels et de formats de données correspondants. Savez-vous dans quelles conditions vous avez le droit d'utiliser ces logiciels ?

Comment représenter de manière simple le code génétique et les 20 protéines ? Le génome humain comporte environ un milliard de bases A/C/G ou T. Combien de disquettes (capacité 1.44Mo) faudrait-il pour sauvegarder votre génome ? Combien de CD (capacité 650Mo) ?

Comment représenter une date ? le pourcentage d'humidité dans l'air ? Comment représenter une coordonnée géographique ? Comment représenter la liste des étudiants de ce module ?

2 TP 1

2.1 Quelques commandes Unix.

Si vous n'avez pas d'interpréteur de commande, il faut utiliser l'interface graphique pour en lancer une :

- Si vous êtes loggué sur une machine CARISM, en appuyant sur le bouton droit de la souris sur le fond de l'écran, et selectionner "**xterm**".
- Si vous êtes sur un PC Linux avec KDE, en cliquant sur l'icone représentant une fenêtre terminal ou en cliquant avec le bouton droit, puis `Executer une commande` puis tapez `xterm` puis `Entree`
- Sous Windows, installez `cygwin` si ce n'est pas fait, dans le menu démarrer choisissez alors CYGWIN ou cliquez sur l'icone de Cygwin.

Attention, les conventions des interfaces graphiques Unix peuvent différer des interfaces que vous connaissez : par exemple, on utilise le click avec le bouton du milieu (ou click simultané sur les boutons droits et gauche) pour insérer le contenu du "presse-papier".

2.1.1 Les répertoires :

- `pwd` : afficher le nom du répertoire où vous êtes (le répertoire courant).
- `~` : désigne votre répertoire d'origine.
  `~miasn1` : désigne le répertoire d'origine de l'utilisateur dont le nom de login est `miasn1`
- `ls` : afficher la liste des fichiers et des sous répertoires du répertoire courant. Pour avoir plus de détails, `ls -l`.
- `cd` : changer de répertoire.
  `cd rep` : pour aller dans le sous répertoire appelé `rep` ;
  `cd` ou `cd ~` : pour aller dans votre répertoire d'origine.
  `cd ~miasn1` : pour aller dans le répertoire d'origine de l'utilisateur `miasn1`.
  `cd ..` : pour aller dans le répertoire parent.
  `cd /usr/local` : pour aller dans le répertoire `/usr/local`
- `mkdir` : Pour créer un sous répertoire, par exemple
  `mkdir rep` crée un sous-répertoire appelé `rep`
- `rmdir` : effacer un répertoire vide, par exemple `rmdir rep`.

Exercice :

A l'aide des commandes expliquées ci-dessus :

1. Placez vous dans votre répertoire d'origine : **cd**. Vérifiez que vous y êtes bien : **pwd**. Regardez la liste des fichiers par : **ls** , puis avec : **ls -al**
2. En utilisant la commande **mkdir**, créer un répertoire de nom : **essai** Vérifier son existence (avec **ls**).
3. Aller dans ce répertoire : **cd essai** Vérifier que vous y êtes (avec **pwd**).
4. Revenir dans le répertoire d'origine par **cd ..** Vérifiez que vous y etes. Détruire (`rmdir`) le répertoire **essai.**

2.1.2 Les fichiers

- `cp` : Copier un fichier :
  `cp essai.cc test.cc` : crée une copie du fichier `essai.cc` nommée `test.cc` (dans le répertoire courant)
  `cp fichier1.cc ..` : copie le fichier1.cc dans le répertoire parent (situé juste avant le répertoire courant).
- `mv` : Déplace ou/et renomme un fichier.
  `mv fichier1.cc fichier2.cc` : renomme le fichier `fichier1.cc`
  Si `rep` est un répertoire, `mv fichier1.cc rep` déplace le fichier1.cc dans le sous répertoire `rep`.
- `rm` : effacer un fichier
  `rm fichier.cc` : efface le fichier `fichier.cc`.
  `rm -r rep` : efface le répertoire `rep` et tous ses sous-répertoires
- `emacs` : éditer un fichier texte
  `emacs fichier.cc &` : ouvre le fichier `fichier.cc` (le crée s'il n'existe pas).
- `mcopy` (en local uniquement, cf. remarque ci-dessous) : permet de copier des fichiers sur une disquette, par exemple :
  `mcopy Programmes/essai.cc a:`
  recopie le fichier `essai.cc` du répertoire `Programmes` sur la disquette. Inversement :
  `mcopy a:essai.cc Programmes`
  recopie le fichier `essai.cc` de la disquette vers le répertoire `Programmes`.
- `mdir a:` (en local) : liste les fichiers contenus sur la disquette

Remarques :

1. Notez que l'appui sur la touche de tabulation (à gauche de la touche A sur un clavier français) est un raccourci clavier qui permet de compléter automatiquement le début d'un nom de fichier ou d'afficher les différentes possibilités de complétion.
2. Le caractère `&` placé à la fin d'une commande permet de l'exécuter en tâche de fonds.
3. La lettre `*` à une signification spéciale : elle permet de remplacer n'importe quel morceau de chaine de caractères, par exemple `ls m*.cc` listera tous les fichiers dont le nom commence par `m` et se termine par `.cc`

4. Pour avoir plus de détails sur une commande Unix, vous pouvez utiliser la commande `man`, par exemple, taper dans une fenetre de commande :
   `man ls`
   pour en savoir plus sur la commande `ls`.
5. Les commandes de copie sur disquette se lancent en local uniquement. Pour se logguer en local, taper simultanément sur les touches `Ctrl Alt` et `F1`, entrez votre login et mot de passe et faites les manipulations sur disquettes, délogguez-vous en tapant simultanément sur les touches `Ctrl` et `d`. Pour revenir à votre session, tapez simultanément sur les touches `Ctrl Alt` et `F7`.

Exercice :
Créez un fichier appelé `essai` contenant quelques lignes. Déconnectez-vous et laissez votre binome se connecter. Recopiez le fichier `essai` de votre binome dans votre répertoire. Renommez-le `essai.de.mon.binome`. Allez dans la salle 11 et recopiez le fichier `essai` sur une disquette.

2.2 Noms et fonction de quelques logiciels.

Ces noms de commande vous permettront de lancer les logiciels correspondants depuis n'importe quel interpréteur de commande (ce qui permet de les lancer indépendemment de l'environnement, il suffit de connaitre leur nom et non pas dans quel menu ils se trouvent).

- `emacs` : éditeur de texte.
- `firefox` : navigateur Internet. Pour utiliser le courrier électronique avec netscape, il vous faudra configurer votre adresse électronique. Permet également de lire des fichiers `.html`
- `abiword` : traitement de texte non scientifique
- `texmacs` ou `lyx` : traitement de texte scientifique particulièrement adapté à la saisie d'équations. Attention à ne pas confondre éditeur de texte et traitement de texte on n'utilise pas `lyx` pour écrire un programme en C++ (cf. section 3.1).
- `gnumeric` (GNU/Linux) : tableur, permet par exemple de lire des fichiers `.xls`
- `xpdf` ou `acroread` : pour lire des fichiers PDF (d'extension `.pdf`)
- `xdvi` : visualiser des fichiers DVI (d'extension `.dvi`)
- `gv` ou `ghostview` : visualiser des fichiers Postscript (d'extension `.ps`)
- `gimp` (GNU/Linux) : traitement d'images par exemple de fichiers d'extension `.gif`, `.png`
- `lp` ou `lpr` : imprime un fichier.
- `zip/unzip` (GNU/Linux) : utilitaires de compression/décompression d'archives au format zip, très répandu sur les machines Windows.
- `scilab` : logiciels de calcul numérique
- `xcas` : logiciel de calcul formel et géométrie interactive
- etc.

3 Les logiciels nécessaires pour programmer.

Rappelons que pour l'ordinateur, un programme est une suite de nombres en mémoire et interprétés par le microprocesseur, c'est le code exécutable. Ce codage n'est évidemment pas adapté à l'être humain. La suite des instructions à faire exécuter par la machine est donc écrite sous forme d'un texte respectant des règles de syntaxe d'un langage, ici le C++, cette suite d'instructions constitue le code source du programme. La traduction du code source en code exécutable est réalisé par le compilateur. La réalisation d'un programme se fait donc en trois temps : écrire le texte constituant le code source, le compiler, puis le tester.

3.1 Écrire le code source avec emacs.

`emacs` est un logiciels appelé *éditeur de texte*, car il permet de créer ou modifier un fichier qui contient du texte. Vous connaissez peut être déjà d'autres éditeurs de texte par exemple : `notepad` ou `wordpad` sous Windows ; `nedit`, `vi` sous Unix, ... Ces éditeurs vont nous servir à écrire le code source (en C++) des programmes.

3.1.1 Lancer emacs

Dans une fenêtre Terminal (*xterm*), taper :

emacs bonjour.cc &

On rappelle que :

- `bonjour.cc` est le nom du fichier C++
- le signe **&** permet de laisser la fenêtre *xterm* disponible pour d'autres commandes UNIX. (Si vous avez oublié le signe `&`, tapez simultanément sur les touches `Ctrl` et `Z` puis tapez `bg`).

Pour faciliter l'utilisation d'emacs, vous pouvez définir des raccourcis claviers, pour cela téléchargez le fichier
`www-fourier.ujf-grenoble.fr/~parisse/.emacs`
et sauvegardez-le dans votre répertoire "home" ou ouvrez le fichier `~/.emacs` et ajoutez-y les lignes suivantes :

```
(setq inhibit-splash-screen t) ; pas d'ecran d'accueil
(show-paren-mode t) ; correspondances de parentheses
(require 'saveplace)
(setq-default save-place t); sauvegarde position du curseur
(global-set-key "\C-g" 'goto-line)
(global-set-key "\C-z" 'advertised-undo)
(global-set-key "\C-v" 'yank)
(global-set-key "\C-c" 'kill-ring-save)
(global-set-key [f1] 'info)
(global-set-key [f2] 'save-buffer)
(global-set-key [f3] 'find-file)
(global-set-key [f9] 'compile)
```

3.1.2 Quelques commandes d'édition utiles.

Pensez à sauvegardez votre texte de temps en temps en utilisant la commande `Save Buffer` du menu `Files` (raccourci clavier `Ctrl-X Ctrl-S` ou `F2`).

Remarque : si vous souhaitez sauvegarder votre fichier source sous un autre nom, vous pouvez utiliser la commande `Save Buffer As` du menu `Files`.

Pour marquer une sélection : cliquer avec le bouton gauche de la souris au début, avec le bouton droit à la fin. Un double-click avec le bouton droit supprime la sélection. Le bouton du milieu permet d'insérer la sélection à la position de la souris. Si votre souris ne possède que deux boutons, cliquez sur les deux boutons simultanément pour insérer la sélection.

L'appui simultané sur les touches `Ctrl` et `Z` permet d'annuler la commande d'édition précédente. L'appui simultané sur `Ctrl` et `G` permet d'interrompre une commande en cours.

L'appui sur la touche de tabulation à gauche de la touche A permet d'indenter le texte source ce qui facilite sa lecture.

Entrez maintenant le texte suivant :

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

int main(){
  cout << "Bonjour! Ca va?" << endl;
}
```

3.2 Compiler votre programme.

Dans la fenêtre de l'interpréteur de commande, tapez :
`g++ bonjour.cc`
Si vous n'avez pas fait d'erreurs le programme sera compilé, c'est-à-dire qu'un fichier exécutable nommé `a.out` sera créé dans le répertoire courant. Vérifiez avec la commande `ls`. Vous pouvez exécuter votre programme en tapant :
`./a.out`
dans la fenêtre de l'interpréteur de commandes.

3.2.1 Quelques options du compilateur g++

Comme pour les autres programme GNU, l'aide en ligne de `g++` est accesible en tapant l'une des commandes :
`man g++`
`info g++`
ou à l'intérieur d'`emacs` en tapant `F1` (ou simultanément sur les touches `Ctrl` et `H` puis en tapant sur `I`). Voici les options les plus utiles :

- L'option `-g` inclut les informations utiles pour la mise au point du programme (avec le deboggueur `gdb`).
- Lorsque votre programme est au point, l'option `-O2` permet d'optimiser votre programme (le rendre plus rapide)

– L'option `-o` suivi d'un nom de fichier permet de donner un nom au programme compilé différent de `a.out`.

3.2.2 Faire compiler emacs.

`emacs` est un éditeur de texte adapté pour faire de la programmation : il affiche par exemple les *mots-clefs* du langage dans une couleur différente, indique la parenthèse ouvrante correspondant à une parenthèse fermante, etc. On peut aussi lancer la commande compilation du texte source dans `emacs`. Pour cela il faut indiquer à `emacs` dans la première ligne du texte source la commande de compilation que l'on exécute habituellement dans la fenêtre de l'interpréteur de commande, par exemple :
`// -*- compile-command: "g++ -g  bonjour.cc" -*-`
Tapez cette ligne dans votre texte source.
Attention, `emacs` n'interprète cette ligne que lorsqu'il ouvre le fichier source. Quittez `emacs` puis relancez `emacs` en tapant :
`emacs bonjour.cc &`

Vous pouvez maintenant lancer la compilation en choisissant `Compile` du menu `Tools` ou en tapant sur la touche `F9` puis validez en tapant la touche `Entree`. La fenêtre `emacs` se divise alors en deux. La partie inférieure indique au fur et à mesure l'évolution de la compilation et les erreurs rencontrées. S'il n'y a pas d'erreur, vous verrez un message ressemblant à :
`Compilation finished at ...`

S'il y a des erreurs, cliquez avec le bouton du milieu sur la ligne de l'erreur dans la fenêtre de compilation, cela placera le curseur au bon endroit dans la fenêtre contenant le texte source. Modifiez alors le texte source et recommencez pour chaque erreur rencontrée. Recompilez jusqu'à ce que tout se passe bien.

Remarques :

– si votre fenêtre `emacs` est déjà coupée en deux morceaux, veillez à taper sur `F9` après avoir sélectionné la demi-fenêtre contenant le texte source.
– si vous avez modifié le nom du texte source avec la commande `Save Buffer As` du menu `Files`, vous devez aussi modifier la première ligne du texte source pour refléter ce changement, puis fermer et réouvrir le fichier pour rendre le changement effectif.[4]
– Si vous lancez la compilation depuis la ligne de commande, dans `emacs`, l'appui sur les touches `Ctrl` et `G` suivi d'un nombre permet de se déplacer aux numéros de lignes d'erreur indiqués par le compilateur.

3.3 Exécuter votre programme.

Si la compilation s'est terminée avec succès, vous pouvez tester le résultat en tapant dans la fenêtre `xterm` :
`./a.out`

[4] Vous pouvez également changer la commande de compilation qui s'inscrit en bas de la fenêtre après avoir appuyé sur F9 et avant d'appuyer sur la touche Entrée

Si votre programme donne des résultats inattendus, vous devrez corriger le texte source et compiler à nouveau. Si tout va bien et que vous souhaitez conserver votre exécutable, vous pouvez le renommer, par exemple, tapez dans la fenêtre `xterm` la commande :

```
mv a.out bonjour
```

Remarque : pour diminuer la taille du fichier exécutable (pour économiser l'espace occupé sur le disque dur), vous pouvez utiliser la commande :

```
strip bonjour
```

3.4 Autres logiciels utiles

- Pour corriger les erreurs un peu récalcitrantes, on utilise un *debuggueur* (logiciel de mise au point), ici `gdb`, ce logiciel sera présenté ultérieurement.
- Certaines erreurs d'allocation mémoire (souvent les plus difficiles à détecter car pouvant provoquer des erreurs de segmentation bien après) nécessitent une exécution sous contrôle d'outils dédiés, comme par exemple `valgrind`.
- Enfin, pour améliorer l'efficacité d'un programme, on pourra utiliser un *profiler*, par exemple `gprof`.

3.5 Ce qu'il faut absolument retenir :

- les commandes Unix de base `ls cp rm mv cd`,
- les commandes d'édition les plus courantes d'`emacs` (section 3.1.2)
- l'enchainement pour créer un programme : édition du source, compilation, exécution du programme compilé (section 3).

4 En complément

Pour installer un compilateur C++ libre sur votre ordinateur :

- PC Windows : installez cygwin `http://www.cygwin.com/`,
- PC Linux : installez les packages ou équivalents `gcc linux-libc-dev glibc-doc`, `g++ libstdc++-dev`, `emacs`, `gdb`
- Mac : installer Xcode, en principe fourni avec le système Mac OS X.

Il existe aussi des compilateurs C++ commerciaux, l'environnement de développement sera alors différent mais bien entendu le langage est identique.

Pour de la documentation en ligne sur C++, un bon point de départ est : `http://www.cplusplus.com/`. Vous pouvez aussi consulter des livres en bibliothèque, par exemple *Le langage C++* de Bjarne Stroustrup (le concepteur du langage).

Pour la documentation de la librairie C standard `libc`, vous pouvez utiliser l'aide d'`emacs` (tapez la touche `F1` dans la fenêtre `emacs`), cherchez et cliquez sur `libc`.
Pour la documentation de la librairie standard C++ (STL), cf. :
`http://www.sgi.com/tech/stl/`

5 Premiers pas en C++.

5.1 Les données.

Tout logiciel manipule des données, celles-ci peuvent être de deux types :
- les constantes, par exemple $\pi = 3.1415...$
- les variables, par exemple le solde d'un compte en banque.

On a vu que pour l'ordinateur ces données étaient une suite de nombres placées à des positions numérotées dans la mémoire appelées *adresses*. Il est peu commode de désigner des données par leur numéro de position en mémoire, les langages de programmation utilisent à la place un nom de *variable* ou de *constante*, c'est le compilateur qui se charge de traduire ce nom en une adresse. Par exemple on pourrait donner `demi` comme nom à la constante mathématique 0.5. Cette valeur ne changera pas, il s'agit d'une constante. On peut choisir le nom `solde` pour la valeur d'un compte en banque, cette valeur change, il s'agit d'une variable. D'autre part, un programme d'ordinateur manipule des données non numériques, par exemple des chaines de caractères en utilisant un format de numérisation des données. Ce format de donnée est dans le langage C++ le *type* de la variable ou de la constante considérée. Il existe des standards de numérisation pour les données manipulées couramment qui correspondent aux types prédéfinis du langage, par exemple :
- `int` pour un entier signé (32 bits)
- `double` pour un nombre réel (en double précision)
- `char` pour un caractère,
- `string` pour une chaine de caractère,
- `vector<int>` pour une liste d'entiers,
- `vector<double>` pour une liste de réels,
- etc.

Lorsqu'on veut utiliser une donnée dans un programme, on *déclare* une variable ou une constante d'un certain type, en indiquant le type de la variable suivi par son nom, par exemple :

```
const double hbar=6.626e-34;
char c ;
...
```

Pour distinguer les variables des constantes, on rajoute le mot-clef `const` (ici on déclare `hbar` comme une constante). On peut aussi *initialiser* une variable lors de sa déclaration ce qui indique sa valeur lors de sa création. C'est indispensable pour une constante.

On peut ensuite utiliser ce nom de variable ou de constante dans la suite du texte source, par exemple, si `R` est une variable réelle représentant le rayon d'un cercle et si on veut déterminer la valeur de la variable réelle `perimetre` représentant son périmètre, on pourra écrire :

```
perimetre=2*M_PI*R ;
```

ce qui calcule le membre de gauche $2\pi R$ et place le résultat dans la variable `perimetre`.

5.2 Exercice

Voici un extrait de programme

```
int i,j=0;
k=i+j;
double l=1.7;
int m=2.3;
double n=2;
ch='A';
char ch;
const double alpha=1.3;
alpha=2.7;
```

Quel est le type des variables `i,j,k,l,m,n,ch` ? A-t-on déclaré toutes les variables ? Quelles sont les variables initialisées ? Les initialisations vous paraissent-elles correctes ?

5.3 Les instructions.

Un programme est une suite d'instructions que l'ordinateur exécute l'une après l'autre. Une instruction en C++ se termine par le signe `;`
Un groupement d'instructions commence par le signe `{` et se termine par le signe `}`

On peut intervenir sur l'ordre d'exécution des instructions en utilisant des commandes de contrôle de l'exécution :

- pour exécuter plusieurs fois le même groupe d'instructions en exécutant une *boucle* tant qu'une condition est vérifiée qui se traduit en C++ par un `for (;;) { }`, par exemple pour calculer la somme des entiers de 1 à 10 :
  ```
  int i=1,somme=0;
  for (;i<=10;i=i+1){
    somme=somme+i;
  }
  ```
  l'instruction ajouter `i` à la variable `somme` est exécutée tant que `i` est inférieur ou égal à 10 et à chaque exécution de la boucle `i` est augmenté de 1.
- pour n'exécuter une instruction que lorsqu'une condition est vérifiée avec un test `if () { } else { }`, par exemple pour le calcul du périmètre de la section précédente, on va tester que le rayon du cercle est positif et afficher un message d'erreur si ce n'est pas le cas :
  ```
  if (R<0){
    cout << "Erreur: le rayon du cercle est negatif!" << endl;
  }
  else {
    perimetre=2*M_PI*R;
  }
  ```

Lorsqu'on souhaite regrouper plusieurs instructions qui permettent de déterminer une donnée en fonction d'autres données, on utilise un autre type de regroupement appelé `fonction`. Par exemple, on peut calculer le nombre de caractères d'une chaine

de caractères. Comme en mathématique, on parle d'argument(s) et de résultat ou valeur de retour de la fonction, dans l'exemple, l'argument est la chaine de caractère et la valeur de retour est le nombre de caractères. On donne alors un nom de fonction (par exemple `taille`) pour pouvoir utiliser cette fonction plus tard dans le texte source. On précise le type des arguments et de la valeur de retour, dans l'exemple :

```
int taille(string s){ ... }
```

Autre exemple :

```
double surface_du_cercle(double rayon){
  double pi=3.14159;
  return pi*rayon*rayon;
}
```

Ensuite on peut utiliser la fonction : `double R=1.3;`

```
double S=surface_du_cercle(R);
```

Exercice :

Écrire une boucle qui calcule le produit des entiers pairs de 2 à 10.
Écrire un test pour déterminer le plus grand de deux entiers. Écrire un test pour vérifier qu'un caractère peut représenter une base du code ADN (pour tester une égalité on utilise ==).
Écrire une fonction qui calcule le volume du sphère en fonction de son rayon. Écrire une fonction qui renvoie le caractère complémentaire d'une base d'ADN (appariés par A-T ou C-G).

5.4 TP : Un programme plus complet.

5.4.1 Le programme

Avant tout nous allons créer un répertoire de travail nommé "Programmes" ; il suffit pour ce faire d'utiliser la commande `mkdir` (make directory) en tapant `mkdir Programmes`.

Dans une fenêtre `xterm` placez vous dans le répertoire souhaité— `cd Programmes`— et tapez la commande

```
emacs essai.cc &.
```

L'éditeur de texte `emacs` s'ouvre alors dans une fenêtre et vous pouvez commencer à taper votre texte **en dessous** des lignes déjà présentes.

Recopiez le texte suivant en utilisant la touche de tabulation après chaque passage à la ligne (de façon à obtenir une indentation automatique).

```
// Fonction qui calcule le carre d'un entier
int carre(int n){
  return n*n;
}

int main(){
  int i;
  for (i=0; i<10; i=i+1){
```

```
    cout << carre(i) << endl;
  }
  return(0);
}
```

Sauvegardez votre texte, tapez sur la touche `F9` puis sur `Entrée` pour lancer la compilation de votre programme. La fenêtre se scinde en deux parties ; l'une contient votre programme, l'autre vous montre l'évolution de la compilation et les erreurs éventuelles. En cas de succès la compilation se termine par le message suivant
`Compilation finished at ...` .
Vous avez obtenu un fichier exécutable, nommé `a.out`, dans votre répertoire de travail. Revenez dans la fenêtre initiale `xterm` et lancez la commande
`a.out` .
Vous devriez voir apparaître la liste des carrés des entiers de 0 à 9.

5.4.2 Explications.

Rappelons que pour l'ordinateur, le programme est une suite d'instructions qui manipulent des données (par exemple des nombres, ou des caractères) dont la valeur est stockée en mémoire. Du point de vue du programmeur, une donnée stockée en mémoire est une variable ou une constante possèdant un type qui décrit le type de données.

Notre programme ci-dessus utilise deux variables `i` et `n` qui sont des nombres entiers, type appelé `int` en langage C. Pour utiliser une variable dans un programme C, il faut la déclarer, c'est-à-dire donner son type et le nom qu'on utilisera ultérieurement pour s'y référer (ce nom s'appelle l'identificateur). Observez les déclarations des variables `i` et `n` : elle se fait en commençant par le type `int` puis l'identificateur (le nom de la variable) puis le signe `,`, le signe `;` ou le signe `)`.

Remarquez que les lignes `int carre(int n){` et `int main(){` ne sont pas des déclarations de variables `carre` et `main`, il y a bien le type `int` et l'identificateur, mais il n'est pas suivi de `;`, `,` ou `)`, il est suivi de `(`. Il s'agit en fait d'une déclaration de *fonction*. Une fonction sert à regrouper différentes instructions simples permettant d'effectuer une opération plus élaborée. Ces instructions sont regroupées à l'intérieur d'un *bloc* délimité par les signes { et }. Comme en mathématiques, une fonction C renvoie un résultat calculé à partir d'une ou plusieurs variables, ce sont les *arguments* de la fonction. Le type du résultat est le type qui précède le nom de la fonction.

Notre programme contient deux fonctions : la fonction `carre` et la fonction `main`, toutes les deux renvoient un nombre entier (type `int`). La fonction `carre` sert (comme son nom l'indique) à calculer le carré d'un nombre entier (c'est ce qu'explique la ligne
`// Fonction qui calcule le carre d'un entier`
toute ligne qui commence par `//` est un commentaire, c'est-à-dire que le texte qui suit est là pour rendre le texte source plus facile à lire par un programmeur mais est ignorè par le compilateur). La fonction `carre` possède un argument de type `int`, qui est l'entier dont il faut calculer le carré. La déclaration des arguments se fait entre les parenthèses qui suivent l'identificateur de la fonction : on indique d'abord le type puis le nom de la variable, s'il y a plusieurs arguments, on les sépare par des virgules. Le nom de variable donné par cette déclaration d'arguments pourra être utilisé à l'intérieur

de la fonction et seulement à l'intérieur de la fonction. Dans l'exemple de `carre`, la variable `n` n'a de sens qu'à l'intérieur du bloc définissant `carre` délimité par le signe { qui suit `int carre(int n)` et le signe } correspondant.

Une fonction doit renvoyer un résultat, pour cela on utilise l'instruction `return` suivi du résultat à renvoyer. Pour la fonction `carre` il s'agit de `n*n`.

Pour utiliser une fonction dans une autre partie du programme, on écrit dans le texte source le nom de la fonction suivi entre parenthèses par les noms de variables que l'on souhaite passer en argument (s'il y a plusieurs arguments on les sépare par des virgules), par exemple ici `carre(i)` dans la fonction `main` sert à calculer le carré de l'entier `i`. Lors de l'exécution de `carre(i)` la valeur de `i` est recopiée dans la variable `n` de la fonction `carre`, la fonction `carre` s'exécute et le résultat est renvoyé pour être utilisé par `cout <<` .

Nous avons fait à peu près le tour de cet exemple, il ne reste qu'à expliquer `for (i=0;i<10;i=i+1)`. Il s'agit d'un exemple de *boucle*, c'est-à-dire que la variable `i` est d'abord initialisée à 0, le bloc délimité par { et } est exécuté avec `i=0`, puis la valeur de `i` est augmentée de 1 (`i=i+1`) et comparée à 10 (`i<10`), comme 1 est inférieur à 10, la boucle s'exécute à nouveau avec `i=1`, `i` est à nouveau incrémenté de 1, etc. jusqu'à ce que `i` vaille 10, la condition `i<10` n'est alors plus vérifiée et l'exécution de la boucle s'arrête, le programme continue après le bloc définissant le `for`.

Remarques :

- La fonction `main` est une fonction un peu spéciale : c'est la fonction qui est exécutée lorsqu'on lance le programme en tapant `a.out` dans la fenêtre `xterm`. Par convention, elle renvoie un résultat de type `int`, on renvoie 0 pour indiquer que le programme s'est déroulé normalement. Pour simplifier on supposera toujours que `main` n'a pas d'arguments.
- Une instruction C peut faire appel soit à des fonctions définies plus haut dans le même programme, soit à des instructions de base, par exemple les opérations sur les nombres entiers +, −, ⋆, etc.
- La fonction `cout <<` permet d'afficher à l'écran la valeur de la variable qui suit <<. Par exemple `cout << carre(i)` affiche la valeur du résultat de `carre(i)`. `endl` permet d'afficher un saut de ligne. `cout` n'est pas une instruction de base, elle fait partie d'une bibliothèque de fonctions appelée `iostream` (i pour input, entrée en anglais, o pour output, sortie en anglais, et stream signifie flux, cette bibliothèque contient des fonctions pour lire des données au clavier et afficher des données à l'écran). Il faut indiquer au compilateur qu'on va utiliser cette bibliothèque de fonctions et c'est le rôle de la ligne :
  `#include <iostream>`
  Il existe de nombreuses bibliothèques de fonctions pour différents domaines d'application, par exemple `string` pour manipuler des chaines de caractères, `vector` pour manipuler des vecteurs, pour les utiliser on place une ligne analogue dans le texte source du programme.

5.5 Exercices

- Modifiez le nom de l'argument de la fonction `carre` (utilisez par exemple `m`)

– Créez une fonction `cube` qui calcule le cube d'un entier et modifiez le programme pour qu'il calcule le cube des entiers de 0 à 9.
– Modifiez le programme pour qu'il calcule les cubes des entiers de 0 à 100. N'oubliez pas de rajouter un commentaire !
– Rajoutez `//` à la ligne `#include <iostream>` et observez le résultat de la compilation.

6 Correction des erreurs détectées à la compilation.

La première condition pour pouvoir compiler correctement, c'est d'appeler le compilateur ! La première ligne de votre fichier doit ressembler à :
`// -*- mode:C++; compile-command:"g++ essai.cc"`
Cette ligne permet à `emacs` de savoir quoi faire lorsque vous lancez la commande de compilation (en tapant sur la touche F9 ou en choisissant `Compile` du menu `Tools`). En particulier, notez que le fichier qui sera compilé est le nom de fichier source qui apparait sur cette ligne, il faut donc toujours s'assurer que ce nom est bien le nom du fichier que vous éditez (ce nom apparait en inverse vidéo sur l'avant dernière ligne de la fenêtre `emacs`). Mais il arrivera certainement un jour ou l'autre que ces deux noms diffèrent (vous devrez alors modifiez la première ligne) ou qu'il n'y ait tout simplement pas de première ligne. Nous commencerons par décrire avec un exemple ce qu'il faut faire dans ce cas. Une fois que l'on a appelé correctement le compilateur, il faut aussi que le texte source soit syntaxiquement correct pour que la compilation aboutisse. Si ce n'est pas le cas, il faut interpréter les messages d'erreur de la compilation, corriger le texte source et essayer de compiler à nouveau. Nous allons également voir comment faire.

Sans quitter Emacs, vous allez maintenant éditer votre second programme que nous appellerons `faux.cc` : à l'aide de la souris sélectionnez "Open file..." du menu "File". Vous devez alors voir apparaitre à la dernière ligne de la fenêtre emacs `Find file:~/`, tapez alors `faux.cc`, le nom du fichier à ouvrir, dans cette ligne de commande. Notez que, puisque vous n'avez pas utilisé la commande `editcc`, votre nouveau fichier est vide. Tapez alors le texte suivant :

```
int carre (int n){
  return n*n ;
}

int main(){
  for(int p=0 ; p<10 ; p++){
    cout << carre(i)<<endl
      int k ;
    5=k ;
    cout << k ;
  }
}
```

Lancez la compilation en tapant `F9`. Le message suivant s'affiche :

`compile command : make -k`
Remplacez `make -k` par :
`g++ faux.cc -lm`
pour lancer la compilation. Vous obtenez les messages :

```
faux.cc : In function 'int main(...)' :
faux.cc :8 : 'cout' undeclared (first use this function)
faux.cc :8 : (Each undeclared identifier is reported only once
faux.cc :8 : for each function it appears in.)
faux.cc :8 : 'i' undeclared (first use this function)
faux.cc :9 : 'endl' undeclared (first use this function)
faux.cc :10 : parse error before 'int'
faux.cc :10 : non-lvalue in assignment
faux.cc :11 : confused by earlier errors, bailing out
Compilation exited abnormally with code 1 at ....
```

Si vous placez votre souris sur la ligne `faux.cc :8` et que vous cliquez avec le bouton du milieu, le curseur se placera automatiquement sur la ligne correspondante du texte source (*i.e.* dans le fichier `faux.cc`). La variable `i` n'est pas **déclarée**, nous nous sommes trompés de nom de variable, nous aurions du écrire :

```
carre(p);
```

La fonction `cout` n'est pas définie non plus, sa définition est faite dans le fichier `iostream` : nous devons donc ajouter la directive

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

en début de fichier (notez que si nous avions lancé l'édition de `faux.cc` en tapant la commande `editcc faux.cc`, cette ligne serait automatiquement écrite en début de fichier). L'erreur suivante provient de l'absence du point virgule après l'instruction `cout << carre(i)` (remarquez à ce propos la mauvaise indentation de la ligne suivante). Rajoutez le `;` déplacez vous sur la ligne suivante et tapez sur la touche de tabulation à la gauche de la lettre `A`, le texte devrait à nouveau s'indenter correctement. La dernière erreur vient de ce que le signe "=" est le signe d'affectation (il sert à copier la valeur du membre de droite vers le membre de gauche donc ici il donne une valeur à une variable) et non le signe utilisé pour tester une égalité. On ne peut affecter la valeur de `k` à une constante (5 ici) : on dit que 5 n'est pas une valeur à gauche (left value).

Une fois les erreurs corrigées la compilation se déroule normalement. Notez alors qu'il y a toujours **un seul** fichier `a.out` dans votre répertoire de travail. Si vous voulez conserver des exécutables (ce n'est pas toujours une très bonne idée car un exécutable occupe beaucoup plus d'espace disponible qu'un fichier source sur le disque dur : comparez la taille de `a.out` et de `faux.cc` (utilisez pour ce faire la commande `ls -l a.out`)), il est nécessaire de les renommer à l'aide de la commande `mv` —move— (`mv a.out carre` par exemple) avant de compiler un autre programme dans le même répertoire.

6.1 En résumé

Bien comprendre les termes suivants : variable, constante, type, déclaration, initialisation, boucle, test, fonction, argument, valeur de retour d'une fonction.

7 Composition d'un programme.

Un texte source écrit en C ou C++ se compose de :

- **Directives du préprocesseur** :
  Ce sont des directives précédées du caractère dièse # qui sont traitées avant la compilation proprement dite. Le préprocesseur modifie le texte source qui sera envoyé au compilateur par exemple en y incluant le contenu de fichiers nécessaires pour utiliser des fonctionnalités définies dans des bibliothèques, ainsi :
  `#include <iostream>`
  est une directive du préprocesseur permettant d'utiliser la bibliothèque pour lire des données au clavier et écrire des données à l'écran. Il existe d'autres type de directives, par exemple pour compiler certaines parties du texte source uniquement dans certains cas, par exemple selon que le système d'exploitation est GNU/Linux ou Windows.
- **Déclarations.** Tout objet (variable, fonction, ...) doit être déclaré avant d'être utilisé. Toute déclaration se termine par un point-virgule.
  La déclaration des variables se fait en indiquant le type de la variable puis le nom de la variable.
  En C++, la déclaration de variables peut être faite n'importe où dans le texte source (mais, bien entendu, avant utilisation). Cela permet de les définir aussi près que possible de l'endroit où elles sont utilisées, ce qui permet plus facilement de relire, corriger ou modifier le code source.
- **Fonctions.** Lorsqu'on doit exécuter à différents endroits d'un programme la même suite d'instructions, on regroupe ces instructions en une **fonction**. Une fonction agit sur des données, qui sont appelées arguments de la fonction. Une fonction renvoie la plupart du temps un résultat. Par exemple la fonction maximum qui calcule le plus grand entier parmi deux entiers a et b a deux arguments entiers a et b et renvoie un entier.
  La déclaration de la fonction est composée du type de la valeur renvoyée, du nom de la fonction, suivi entre parenthèses du type et du nom des arguments de la fonction séparés par des virgules. Le corps d'une fonction (la partie délimitée par le { qui suit la liste des arguments et le } fermant correspondant) est constitué d'instructions. Chaque instruction doit se terminer par un point-virgule.
  Le programme doit contenir une fonction dont le nom est `main`, c'est cette fonction qui est exécutée lorsqu'on tape `a.out` dans la fenêtre `xterm`.
  Certaines fonctions sont rattachées à un type d'objet, par exemple la longueur pour une chaine de caractères ou le nombre d'éléments d'un vecteur, on peut alors les voir comme une caractéristique de l'objet, on parle alors de **méthode**.
- **Commentaires.** Il est souvent nécessaire d'inclure des commentaires pour expliquer à un lecteur éventuel (ou à l'auteur...) le fonctionnement du programme. Un commentaire sur plusieurs lignes commence par le signe `/*` et termine par `*/`. On peut aussi commenter **une** ligne en la faisant débuter par `//`.

exemple :

```
#include <iostream> // directive du preprocesseur
using namespace std;
```

```
// declaration et definition de la fonction max
int max(int a,int b){
  if (a>b)
    retun a;
  else
    return b;
}

// declaration et definition de la fonction main
int main(){
  int n,m; // declaration des variables n et m
  cout << "Donnez deux entiers: " ;
  cin >> n >> m ;
  cout << "Le plus grand des deux est: " << max(n,m) << endl;
}
```

8 Quelques types de variables.

8.1 Les types simples

Toutes les variables ont un **type** qui doit être déclaré sous la forme suivante :
`type nom-de-variable;`
On peut grouper la déclaration de plusieurs variables de même type, par exemple :
`type 1er-nom-de-variable, 2eme-nom-de-variable;`

Les principaux types sont :

char caractère,
Lorsqu'on veut utiliser un caratère fixé, on l'écrit entre apostrophes. Par exemple si `c` est une variable de type `char`, on peut écrire
`c='A';`
pour donner à la variable `c` la valeur `'A'`. Attention à ne pas confondre le nom de la variable (ici `c`) et sa valeur (qui est `'A'`, ce pourrait être le caractère `'c'`). En réalité l'ordinateur stocke un entier compris entre 0 et 255 appelé code ASCII de la lettre. Par exemple le code ASCII de `A` est 65, celui de `B` est 66, etc.

int entier, options : *short, long, signed, unsigned*
(Noter qu'il n'est pas nécessaire de préciser `int` si on indique `short, long, unsigned`).

float nombres à virgule flottante simple précision
Il s'agit de nombres réels, par exemple 3.1415, contrairement aux entiers ci-dessus. Remarquez que le nombre réel 3.1415 ne peut être stocké dans une variable de type `int`. Par contre, un nombre entier comme par exemple 13 peut être stocké sous deux formats : `int` bien sur, mais également `float` (pour éviter les confusions, on le note alors avec un point décimal à la fin `13.`) : dans ce

cas il sera considéré comme un nombre réel ce qui peut changer le résultat de certaines opérations (par exemple la division `/` renvoie le quotient euclidien si les arguments sont entiers et le quotient réel sinon, ainsi `13/2` vaut `6` alors que `13./2.` vaut `6.5`).

double nombres à virgule flottante double précision, option : *long*
Comme `float` mais avec plus de chiffres significatifs.

bool une variable de ce type peut prendre deux valeurs : `false` ou `true`. Une valeur de type entier qui est nulle [respectivement non nulle] est considéré comme `false` [respectivement `true`].

Exercice :
Écrire un programme qui lit un réel et affiche son carré.

Remarque :
Comme dit précédemment un caractère est codé par son code ASCII, qui est un entier compris entre 0 et 255, on peut donc écrire :

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

int main(){
  int i=65 ;
  char ch=i ;                          // 65 est le code ASCII du caractère A
  cout ≪ ch ≪ "=  " ≪ i ≪ endl ;
  ch= 'B' ;
  i=ch ;
  cout ≪ ch ≪ "=  " ≪ i ≪ endl ;
}
```

8.2 Conversion de type.

Il est parfois nécessaire de changer le type d'une variable on parle alors de "casting". Par exemple si p et q sont deux entiers, `p/q` donne le quotient entier de p par q. Si l'on veut au contraire une valeur décimale approchée de p/q, on écrira :
`((float) p)/q;` ou `float(p)/q`.

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

void main(){
  int p,q ;
  p=1 ; q=3 ;
  cout ≪ "comparez " ≪ p/q ≪ " et " ≪ (double) p/q ≪ endl ;
  float x=1.999999 ;
  int i= (int) x ;                     // conversion d'un float en entier
  cout ≪ i ≪endl ;

}
```

Le résultat est le suivant :

```
> a.out
comparez 0 et 0.333333
1
```

8.3 Les vecteurs

Ce type permet de représenter des tableaux dont la taille peut varier. Ce type n'est pas un type de base, il faut donc écrire au début de votre fichier source :

```
#include <vector>
```

Dans la déclaration d'un vecteur, il faut indiquer le type d'élément du vecteur entre crochets, par exemple pour un vecteur de nombres réels :

```
vector<double> v;
```

Dans cet exemple on a déclaré un vecteur qui est vide pour l'instant. On peut aussi déclarer un vecteur ayant n éléments, par exemple :

```
vecteur<int> v(15);
```

déclare un vecteur contenant 15 entiers. On peut enfin créer un vecteur contenant n éléments ayant tous la même valeur, par exemple :

```
vecteur<double> v(10,1.2);
```

crée un vecteur ayant 10 éléments, chaque élément est initialisé à la valeur `1.2`.

La taille d'un vecteur est donné par sa méthode `size()`, par exemple :

```
int s=v.size();
```

Les éléments d'un vecteur sont numérotés de 0 à $s-1$ où s est la taille du vecteur, on accède au i-ième élément du vecteur par `v[i]`.

On peut rajouter un élément à la fin d'un vecteur en utilisant la méthode `push_back`, par exemple si `v` est un `vector<int>` :

```
v.push_back(13);
```

On peut enlever le dernier élément d'un vecteur en utilisant la méthode `pop_back()`.

Les matrices :

Pour définir une matrice, on utilise un vecteur de vecteurs, par exemple

```
vector< vector<double> >
```

est le type d'une matrice de réels. On peut créer une matrice de taille fixée, par exemple 10 lignes et 5 colonnes initialisé à 0, de la manière suivante :

```
vector< vector<double> > M(10,vector<double>(5));
```

On accède ensuite à l'élément ligne i, colonne j de la matrice par `M[i][j]`. Attention, ne pas utiliser `M[i,j]` qui désigne la j-ième ligne de la matrice et non l'élément ligne i colonne j.

Tri d'un vecteur :

On peut trier un vecteur par ordre croissant en utilisant la fonction `sort`. Mettre en début de fichier `#include <algorithm>` puis taper dans le texte source :

```
sort(v.begin(),v.end());
```

pour trier le vecteur `v`. Pour utiliser un autre critère de tri, on définit une fonction prenant en argument deux variables du type des élements du vecteur et renvoyant le type `bool` et on rajoute le nom de la fonction comme troisième argument de la fonction `sort`. Par exemple pour trier un `vector<int>` par ordre décroissant :

```
bool tri(int a,int b){ return a>=b; }
sort(v.begin(),v.end(),tri);
```

Exemple :

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

int main(){
  vector<int> v(10,7); // vecteur a 10 elements valant tous 7
  v[9]=0; // dernier element mis a 0
  cout << "Entrez la valeur du premier element du vecteur: ";
  cin >> v[0] ;
  cout << "Les coordonnees de v sont: ";
  for (int i=0;i<v.size();i++)
    cout << v[i] << " ";
  cout << endl;
}
```

Exercice :
Écrire un programme qui calcule la moyenne d'un vecteur de `double`. Puis en triant le vecteur, déterminer la médiane.

Écrire un programme qui détermine le maximum et le minimum d'un vecteur de `int`. Plus difficile : déterminer la position (l'indice) où le maximum et le minimum sont atteints.

8.4 Les chaines de caractères

Il s'agit d'un type représentant une suite de caractère. Ce type n'est pas un type de base, il faut donc mettre la ligne :
`#include <string>`
au début de votre fichier source.

Le type de variable est `string`, par exemple :
`string s`
déclare une variable `s` de type chaine de caractères. On peut initialiser une chaine de caractère :

- par une chaine écrite dans le texte source délimitée par `"`[5] :
  `string s("Coucou");`
- par une autre chaine de caractère (`string t(s);`) ou un morceau de chaine de caractère (`string t(s,0,3);`) recopie 3 caractères consécutifs de `s` dans `t` en commençant par `s[0]`.
- par la répétition d'un même caractère, par exemple :
  `string s(15,'A');`
  crée la chaine de caractères `"AAAAAAAAAAAAAAA"`.

[5] Attention, on utilise le délimiteur `"` pour les chaines de caractères et `'` pour un caractère

On peut concaténer (mettre bout à bout) deux chaines en utilisant l'opérateur `+`, cet opérateur `+` permet aussi de rajouter un caractère à une chaine de caractères, par exemple :

```
s = s + '.';
```

rajoute un point à `"Coucou"`.

Pour connaitre la longueur d'une chaine de caractère, on utilise sa *méthode* `size()`, par exemple :

```
int l=s.size();
```

Dans une chaine de caractère, les caractères sont numérotés de 0 à $l-1$ où l est la longueur de la chaine. Pour accéder au i-ème caractère de la chaine `s`, on utilise `s[i]` ou `s.at(i)`, par exemple :

```
char c=s[i];
```

Pour accéder à un morceau de la chaine `s` on indique la position de départ et le nombre de caractères, par exemple :

```
string s("Bonjour");
string t=s.substr(3,4); // renvoie "jour"
```

Remarque : en C, le type `string` n'existe pas, on utilise alors le type tableau `char *`, la méthode `c_str()` permet de convertir une chaine de type `string` en un tableau de caractères C, ce qui s'avère parfois nécessaire (par exemple pour convertir des données en chaine de caractères, cf. la section 11.3).

Exercice :

Écrire une boucle calculant le nombre d'apparitions du caractère `'A'` dans une chaine de caractère.

Écrire une boucle affichant les caractères d'une chaine par groupe de 3.

8.5 Autres types.

La librairie standard C++ (STL) permet de créer des structures de données classiques comme les annuaires (`map`), les piles (`stack`), les tas (`heap`), ... Par exemple pour créer un répertoire de numéros de téléphone, on ajoute en début de fichier :

```
#include <map>
#include <string>
```

puis on déclare la variable `tel` qui servira de répertoire :

```
map<string,string> tel;
```

Le premier type paramètre de `map` (`string` ici) sert à indiquer le nom de la personne, le second type indique le numéro de la personne, on a encore choisi le type `string` car le type `int` est trop limité pour contenir des numéros de téléphone de 10 chiffres.

Pour rajouter un numéro :

```
tel["Gaston"]="0476000000";
```

Pour afficher le numéro de Gaston :

```
cout << tel["Gaston"] << endl;
```

Pour chercher si Gaston possède un numéro on écrit :

```
if (tel.find("Gaston")!=tel.end())
```

Pour afficher tous les numéros du répertoire :

```
map<string,string>::const_iterator it=tel.begin(),itend=tel.end();
```

```
for (;it!=itend;++it)
  cout << it->first << " : " << it->second << endl;
```

8.6 TP 3

Écrire un programme qui lit trois nombres et retourne le maximum.

Créer un vecteur contenant les carrés des entiers de 10 à 19, demandez à l'utilisateur d'entrer un entier i (entre 0 et 9), affichez le i-ième élément du vecteur.

Écrire un programme renvoyant l'ARN codé par une chaine d'ADN. Le code ADN utilise les lettres A, T, C, G et le code ARN correspondant est obtenu par correspondance de bases A-U, T-A, C-G, G-C.

8.7 Type défini par l'utilisateur (struct)

Il est parfois souhaitable de regrouper en un bloc plusieurs données, par exemple on pourrait créer un type `date` qui regrouperait 3 entiers désignant respectivement le jour, le mois et l'année. On utilise à cet effet le mot-clef `struct` suivi du nom du nouveau type de donnée créé, puis on définit le type, dans notre exemple :

```
struct date {  int jour; int mois; int annee;  };
```

On peut alors déclarer une variable de type `date`, par exemple :
`date naissance;`
`naissance.jour=2; naissance.mois=3; naissance.annee=1980;`
crée la variable `naissance` et lui donne la valeur 2 mars 1980.

Remarques :

- n'oubliez pas le `;` après l'accolade fermante `}` lorsque vous définissez un type `struct`.
- `typedef` permet de donner un nom de type synonyme d'un type existant, par exemple :
  ```
  typedef vector<double> vect;
  ```
- On peut définir un *constructeur* pour faciliter la création de variables d'un type structuré, par exemple pour le type `date` :
  ```
  struct date {
    int jour,mois,annee;
    date(j,m,a):jour(j),mois(m),annee(a) {};
  };
  ```
  on peut alors déclarer une variable `d` valant le 2 janvier 2002 par :
  ```
  date d(2,1,2002);
  ```

Exercice :
Définir une structure pour représenter les coordonnées spatiales d'un point. Ecrire la distance entre deux points en utilisant cette structure.

9 Instructions fondamentales

9.1 Opérations.

affectation

La syntaxe de l'affectation est `x= e` où `x` est une variable (déclarée) et `e` est une expression dont le type est compatible avec celui de `x` (si `x` est un réel (float ou double) `e` peut être entier). L'expression `e` est évaluée et sa valeur est affectée à `x`.
Attention `(n=10)` renvoie la valeur affectée (ici 10), ce qui permet d'écrire :
`m=(n=10)+5`
qui stockera 10 dans `n` et 15 dans `m`.
exemple :

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

int main(){
 int n ;
 if (n=0) cout ≪ "n=0" ≪endl ;                    // piège : (n=0) renvoie la valeur 0
 else cout ≪ "n≠0" ≪endl ;                        // il faut écrire n= = 0
}
```

Opérateurs arithmétique

Les opérateurs arithmétiques suivants sont définis pour les types réels (`int, float, double`) : `+ - * /`

Remarques :

- le résultat d'une division d'entiers est un entier (c'est le quotient euclidien, autrement dit la partie entière du quotient) par exemple `5/3` donne `1`.
- Pour les entiers `a % b` donne le reste de la division de `a` par `b`.
- Contrairement à la notation mathématique, on ne peut pas utiliser `a^b` pour calculer une puissance. Pour les `float` et les `double`, on utilisera `pow(a,b)` (cf. la section 14).

Autres opérateurs

incrémentation : `n++` la valeur de `n` est incrémentée de 1 (dans le cas d'une boucle `for` l'incrémentation se fait **après** passage dans la boucle).
décrémentation : `n-` la valeur de `n` est décrémentée de 1 (dans le cas d'une boucle `for` la décrémentation se fait **après** passage dans la boucle)
ou : ||
et : &&
non : !
égalité : == ($a == b$ est vrai (1) si $a = b$ et faux (0) sinon.)
différent : !=
inférieur ou égal : <=
supérieur ou égal : >=
Attention :
Lorsqu'on teste qu'une variable a est égale à une autre variable b, il faut écrire a==b et non pas a=b qui recopie la valeur de a dans b

9.2 Les tests.

if

Exécute une instruction lorsqu'une condition est vérifiée.

Syntaxe :

```
if (expression) { instruction ou bloc d'instructions }
[ else { instruction ou bloc d'instructions } ].
```

Exemple

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

int main(){
  int an;
  cin >> an;
  if ( (an%4 ==0) && ( (an%100 !=0) || (an%400 ==0) ) ){
    cout <<"année bissextile"<< endl;
  }
  else
    cout << "année non bissextile";
}
```

switch

Exécute une instruction parmi un ensemble d'instructions selon la valeur d'un entier.

syntaxe :

```
switch(type entier){
 case 1 : instruction(s);    break;
 case 2 : instruction(s);    break;
 default : instruction;
}
```

Exemple :

```
// -*- mode :C++ ; compile-command : "g++ -g essai.cc -lm" -*-
// Auteur :
// Date :

#include <iostream>

using namespace std;

int main(){
  double x,y,res=0;
  char op;
  cin >> x >> op >> y;
  switch(op){
```

```
  case '+' :
    res=x+y ;
    break ;
  case '-' :
    res=x-y ;
    break ;
  case '/' :
    if (y !=0)
      res=x/y ;
    else
      cerr << "division par 0" << endl ;
    break ;
  default :
    cerr << "Operation non prevue !" << endl ;
  }
  cout << res << endl ;
  return 0 ;
}
```

On a tout intérêt à remplacer, dans la mesure du possible, une suite de "if"(s) par un `switch` : d'une part c'est plus facile à lire, d'autre part `switch` est plus rapide à l'exécution qu'une succession de "if".

9.3 Les boucles

for

Exécute une instruction tant qu'une condition est vérifiée.

syntaxe :

```
for (initialisation; condition d'arret; incrémentation) {
   instruction; ou bloc d'instructions
}
```

Il est possible de déclarer une variable de boucle dans l'instruction `for` : `for(int i=0;...;...){ }`. Cette variable ne sera cependant pas utilisable hors du bloc de la boucle.

On peut arrêter la boucle à tout moment en utilisant l'instruction `break;`. On peut arrêter l'itération actuelle et passer à l'itération suivante en utilisant l'instruction `continue;`

Exemple :

```
  for (int i=0 ; i< 10 ; i++) cout << setw(5) << i*i*i <<endl ;
```

while

Il s'agit d'un cas simplifié de boucle `for`. Il n'y a pas d'instruction d'initialisation de boucle ni d'instruction d'évolution.

syntaxe :

```
while (expression) { instruction ou bloc d'instructions }
```

Exemple :
Calcul de la factorielle d'un (petit) entier.

```
int facto(int n){
  int j=1 ;
  while (n !=0){
    j=j*n ;
    n-- ;
  }
  return (j) ;
}
```

do ... while

Comme `while`, mais la condition d'arrêt est testée **après** l'exécution du bloc d'instruction de la boucle. La boucle est donc exécutée au moins une fois.
syntaxe :
do { `instruction(s) ;` } **while** `(expression ) ;`.
Exemple :

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

int main(){
  char rep ;
  do {
    cout << "voulez vous jouer (o/n)" <<endl ;
    cin >> rep ;
  }
  while ( (rep !='o') && (rep !='n') ) ;
  if (rep=='o')
    cout << "on n'est pas ici pour s'amuser" <<endl ;
  else
    cout << "au revoir" <<endl ;
}
```

9.4 TP4 : Exercices

(maths–) Écrire un programme qui lit au clavier une durée en secondes et la convertit en jours, heures, minutes et secondes.

(génétique-) Écrire un programme vérifiant que 2 brins d'ADN sont complémentaires (i.e. les lettres des 2 brins se correspondent par la règle A-T, T-A, C-G, G-C)

(maths-) Construire un programme qui lit un entier positif n et calcule les termes successifs de la suite définie par récurrence de la façon suivante : $u_0 = n$, $u_{n+1} = u_n/2$ si u_n est pair , $u_{n+1} = 3u_n + 1$ sinon. On s'arrête lorsque $u_n = 1$ ou lorsque le nombre d'itérations est trop grand ; dans tous les cas on affiche le nombre d'itérations et un message qui permet de distinguer les deux cas.

(maths-) Construire un programme qui lit un entier N et calcule la somme $\sum_1^N \frac{1}{k^2}$ dans le sens des indices croissants puis décroissants.

(maths) Construire un programme qui, étant donnée une droite dans le plan, détermine si deux points du plan sont placés de part et d'autre de la droite ou non.

(génétique) Écrire un programme qui détermine le début et la fin d'une partie codante d'une séquence d'ARN (en recherchant le premier triplet de démarrage AUG et l'un des triplets de fin correspondants UAA, UGA, UAG). Modifier le programme précédent pour qu'il lise de 3 en 3 seulement et détermine les 3 possibilités de partie codante de l'ARN (selon que l'on commence à la 1ère, 2ème ou 3ème lettre la lecture de 3 en 3). On renverra pour chacune la longueur de la séquence codante et le rapport du nombre de bases AU sur le nombre de bases CG.

10 Les fonctions.

On peut définir une fonction de deux manières selon que le bloc d'instruction est accolé à la déclaration du prototype de la fonction ou non. Dans le deuxième cas on déclare la fonction puis on la définit :

Déclaration d'une fonction

```
type Nom-de-fonction(liste-typée-des-arguments);
```

Le type est le type de la valeur prise par la fonction. Le type `void` précise simplement que la fonction ne retourne pas de valeur.

La liste des arguments peut être vide : dans ce cas, la déclaration se fait de la façon suivante :

```
type Nom-de-fonction();
```

Attention :

Ne pas oublier de mettre des parenthèses lors de l'appel de la fonction même si elle ne prend aucun argument.

La déclaration des fonctions, si elle n'est pas toujours nécessaire dans de petits programmes, est indispensable si l'on veut appeler une fonction avant de l'avoir définie ou l'utiliser dans des modules distincts (on parle de "programmation modulaire").

Définition d'une fonction

```
type Nom-de-fonction(liste-typée-des-paramètres)
bloc d'instructions
```

Exemple 1 :

```
#include <iostream>
#using namespace std;
void salut(){
  cout << "salut toi !" <<endl;
}

int main(){
  salut();
```

```
}
```

Exemple 2 :

```
#include <iostream>
#using namespace std;
float carre (float x) ;                          // déclaration de la fonction

int main(){ // declaration et definition de main
   float x ;
   cin ≫ x ;
   cout ≪ " le carré de " ≪ x ≪ "est " ≪ carre(x) ≪ endl ;
}

float carre (float x){                           // définition de la fonction carre
   return (x∗x) ;
}
```

Exemple 3 :

```
#include <iostream>
#using namespace std;
float pointure(float x) ;
void sortie(string nom, float y) ;

int main(){
   cout ≪ "donnez votre nom de login"≪ endl ;
   string nom ;
   cin ≫ nom ;
   cout ≪ "Quelle est la longueur de votre pied en centimètres"≪ endl ;
   float pied ;
   cin ≫ pied ;
   sortie(nom, pied) ;
}

float pointure(float x){
   return (1.5 ∗ (x+1)) ;
}

void sortie(string nom, float y) {
   cout ≪ nom ≪ "votre pointure est " ≪ pointure(y) ≪ endl ;
}
```

Remarques :

- L'instruction `return (y)` permet d'assigner la valeur `y` à valuer renvoyée par la fonction.
- Une instruction `return` provoque une sortie immédiate de la fonction.

– On appelle parfois “procédure” une fonction de type `void` (qui ne comporte donc pas d’instruction `return`).

Exercices :

1/ Écrire une fonction qui renvoie 1 si deux caractères d’un code ADN sont complémentaires et 0 sinon (les couples A T et C G sont complémentaires). Utilisez cette fonction pour écrire une fonction qui renvoie 1 si deux chaines d’ADN sont complémentaires et 0 sinon.

2/ Écrire une fonction qui renvoie la chaine d’ADN complémentaire d’une chaine donnée.

3/ Écrire une fonction qui renvoie le vecteur somme des deux vecteurs passés en argument. Utilisez cette fonction pour écrire une fonction qui renvoie la matrice somme de deux matrices.

4/ Écrire une fonction qui calcule le produit scalaire de deux vecteurs. Écrire une fonction qui calcule la transposée d’une matrice. Utilisez ces deux fonctions pour calculer le produit de deux matrices.

Passage par référence

Une fonction ne peut renvoyer qu’une et une seule valeur. Il arrive parfois qu’on souhaite modifier la valeur de plusieurs variables à l’intérieur d’une même fonction. Par exemple considérons une fonction `virement` qui effectue le virement d’une somme d’un compte bancaire à un autre. Cette fonction doit modifier deux variables en même temps : le solde du compte débité et le solde du compte crédité. Supposons aussi que le compte à débiter soit inscrit dans le deuxième classeur des archives de la banque et que le solde actuel soit de 10000F.

Les fonctions que nous avons vu jusqu’à maintenant recopient les arguments lors de l’appel de la fonction dans de nouvelles variables qui sont définies à l’intérieur de la fonction. Elles peuvent alors modifier les valeurs de ces nouvelles variables mais ces modifications ne se répercutent pas sur la valeur des variables dont elles sont la copie.

L’idée est en fait de passer en arguments à la fonction non pas la valeur d’une variable mais l’adresse mémoire où est stocké le contenu de la variable. Dans l’exemple ci-dessus, au lieu de passer en argument la valeur du solde (c’est-à-dire le nombre 10000F), on passe en argument l’endroit où est écrit la valeur du solde (c’est-à-dire “le nombre du deuxième classeur”)

C’est ce qu’on appelle passer des arguments *par référence*.

Les arguments passés par référence sont précédés du signe `&`.

Dans l’exemple ci-dessous, `& debit` représente en fait l’adresse (en mémoire) de la variable `debit` (c’est-à-dire l’adresse mémoire où est stocké le contenu de la variable)

exemple :

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

void virement(int somme, int & debit, int & credit){
  debit = debit - somme;
  credit = credit + somme ;
  somme = 0;
```

```
}

int main(){
  int solde_client1 = 10000;
  int solde_client2 = 5000;
  int montant;
  cout << "Tapez le montant a virer: ";
  cin >> montant ;
  virement(montant,solde_client1,solde_client2);
  cout << "Montant vire: " << montant << endl;
  cout << "Nouveau solde du client 1: " << solde_client1 << endl;
  cout << "Nouveau solde du client 2: " << solde_client2 << endl;
}
```

Observez que la fonction `virement` modifie les variables `solde_client1` et `solde_client2` mais ne modifie pas la valeur de la variable `montant` qui n'est pas passée par référence (mais a été recopiée dans la variable `somme` de `virement`). A la fin de l'exécution de `virement`, `somme` vaut 0, mais `montant` n'a pas changé.

Exercice :
Modifier la fonction de l'exercice 1/ ci-dessous en ajoutant un argument de type entier qui sera augmenté de 1 si les 2 caractères sont complémentaires. Utilisez cette fonction pour compter le nombre de caractères complémentaires dans deux chaines de caractères.

Écrire une fonction transposée qui modifie son argument en la matrice transposée et ne renvoie rien.

Remarque
On utilise aussi le passage par référence dans des situations où on **ne modifie pas** l'argument. Il s'agit alors d'éviter de recopier des données de grande taille, par exemple un vecteur contenant un million d'éléments. On fait alors précéder le type de la variable par le mot-clef `const`, par exemple :

```
int somme(const vector<int> & v){
  int resultat=0;
  for (int i=0;i<v.size();i++)
    resultat = resultat + v[i] ;
  return resultat;
}
```

Définition inline Un appel d'une fonction de ce type dans le code source n'est pas remplacé par un appel de la fonction dans le programme exécutable, à la place, le code source de la fonction est recopié intégralement (autant de fois que la fonction est appelée). L'exécution est un peu plus rapide mais le code engendré peut devenir beaucoup plus grand.
Exemple

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

inline int cube(int n) { return n*n*n ;}

int main(){
  cout << cube(6)<< endl ;
}
```

Remarque Il est possible de donner des valeurs *par défaut* à certains paramètres (les derniers de la liste des paramètres). Quand on appelle la fonction, on peut donner ou ne pas donner la valeur de ces arguments, dans le deuxième cas les valeurs par défaut sont utilisées.

```
double f(double x, int n=0) ;
double f(double x, int n){
  return (x*x + n*x) ;
}

int main(){
  cout << f(3.0)<<endl ;
  cout << f(3.0 ,5)<<endl ;
}
```

11 Lecture et écriture des données.

Cette section explique comment transférer des données écrites en toutes lettres depuis le clavier, vers l'ecran, depuis et vers des fichiers texte ou depuis et vers des chaines de caractères.

11.1 Lire au clavier et écrire à l'écran.

Trois flots ("stream") sont prédéfinis après inclusion du fichier `iostream` :

cout correspond à la sortie (`out`) standard (écran en général...)

cin correspond à l'entrée (`in`) standard (clavier en général)

cerr est utilisé pour envoyer des messages d'erreur (à l'écran en général).

<< permet d'envoyer des valeurs dans un flot de sortie.
>> lit des données dans un flot d'entrée.
Il faut s'imaginer que les données s'écoulent dans le sens d'un entonnoir >>.

Remarque :
`cout` n'affiche pas forcément les données à l'écran immédiatement. Il place les données dans une zone mémoire transitoire (*buffer* en anglais). Cette zone sera affichée d'un seul coup par exemple lorsqu'on demande de passer à la ligne, pour cela on peut

utiliser l'instruction :
`cout << endl;`
exemple :

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

int main(){
  int a, b;
  cout ≪ " donnez deux entiers a et b (b non nul)"≪endl;
  cin ≫ a ≫ b;
  if (b==0) { cerr ≪ "division par 0" ≪endl; }
  else {
    cout ≪ " le quotient entier de a par b est " ≪ a/b ≪ endl;
    cout ≪ " le reste de a par b est " ≪ a % b ≪ endl;
  }
}
```

Notez le test `if (b==0)` : si b est nul, le bloc délimité par `{` et `}` entre `if` et `else` est exécuté, sinon c'est le bloc qui suit `else` qui est exécuté.

L'inclusion du fichier en-tête `iomanip` permet de controler les paramètres d'affichage. Par exemple :
`precision (int n)`
définit la précision (à l'affichage) des nombres réels (type `float` ou `double`)
`setw (int n)`
définit la taille en nombre de caractères d'un champ d'affichage.
exemple 1 :

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

int main(){
  cout.precision(6);
  cout ≪ sqrt(2);
}
```

donne `1.41421`.

exemple 2 :

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

```
int main(){
  for (int i=0 ; i< 10 ; i++) cout << setw(5) << i*i*i <<endl ;
}
```

Exercice :
Lire un réel et l'afficher successivement avec 0,1,2,3,4,5 et 6 décimales.

11.2 Lire et écrire sur le disque dur.

Jusqu'à présent nous avons lu des données au clavier et écrit des données à l'écran. Nous allons voir maintenant comment lire des données depuis un fichier et écrire des données dans un fichier (sur le disque dur).

Le fichier header à inclure s'appelle `fstream` au lieu de `iostream`. Ensuite il faut déclarer une variable qui fera le lien avec le nom du fichier sur le disque dur. Par exemple supposons que le nom du fichier sur le disque dur s'appelle `fichier_disquedur` et qu'on souhaite utiliser le nom de variable `fichier_prog` dans le texte source.

- Pour lire des données dans ce fichier, on déclare :
  `ifstream fichier_prog("fichier_disquedur");`
  puis on utilise `fichier_prog` comme `cin`, par exemple :
  `fichier_prog >> donnee ;`
- Pour écrire des données dans ce fichier, on déclare :
  `ofstream fichier_prog("fichier_disquedur");`
  puis on utilise `fichier_prog` comme `cout`, par exemple :
  `fichier_prog << donnee << endl ;`

11.3 Lire et écrire dans des `string`.

Cela permet en particulier de convertir des chaines de caractères en des types entiers ou réels et inversement.

Le fichier en-tête à inclure est `#include <strstream>` (vous devrez aussi inclure `#include <string>`). Comme pour les flots fichiers, on définit une variable qui fait le lien entre la chaine de caractère et la variable de flot.

- Pour lire des fichiers depuis une chaine :
```
string s("15 22 13.6 64.1");
istrstream a_lire(s.c_str());
int i1,i2; double d1,d2;
a_lire >> i1 >> i2 >> d1 >> d2;
```
- Pour écrire des données dans une chaine de caractère :
```
ostrstream a_ecrire;
double d=17.3; int i=56;
a_ecrire << d << " " << i ;
string t(a_ecrire.str()); // Place le resultat dans la chaine t
```

12 Le générateur de nombres aléatoires

Vu son importance, nous allons maintenant décrire le générateur de nombre aléatoire de la librarie standard C. La librairie standard C (`#include <cstdlib>`) fournit deux instructions pour renvoyer des nombres (pseudo-)aléatoires : `rand` et `srand`.
`rand()` renvoie un entier compris entre 0 et la constante entière `RAND_MAX`. `srand(i)` sert à initialiser le générateur de nombre aléatoire, pour une valeur de i donnée la suite de nombre renvoyée par des appels successifs à `rand` sera toujours la même ce qui est très utile pour mettre au point un programme. Une fois le programme mis au point, on initialise en général le générateur de manière plus aléatoire, par exemple à l'aide de l'horloge du système.

Exemple :

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

int main(){
  srand(time(NULL)); // initialise le generateur de maniere aleatoire
  for (int i=0 ; i<10;i++ ) {
    float f=rand()/(RAND_MAX+1.0);
    cout << "Voici un reel compris entre 0 et 1: " << f << endl ;
  }
  return(0);
}
```

Notez le `1.0` (et non pas `1`), sinon la division définissant `f` serait une division d'entiers et renverrait toujours 0.

Pour avoir toujours la même suite de nombres, remplacez `srand(time(NULL))` par exemple par `srand(2000)`.

Pour générer un entier aléatoire entre 1 et 10 on utilisera :
`j=1+(int) (10.0*rand()/(RAND_MAX+1.0));`

Exercices : (proba) Simuler un tirage aléatoire de 2 dés (à l'aide des fonctions `rand` et `srand`). Vérifier expérimentalement sur un grand nombre de tirages les fréquences d'apparition de 2 à 12.

(proba) Simuler un tirage d'une chaine de caractère de longueur aléatoire et dont les caractères sont choisis aléatoirement parmi les caractères A, C, G, T.

13 TP 5

Écrire une fonction permettant de rentrer les coefficients d'une matrice de réels.

Écrire une fonction affichant une matrice de nombres réels (de type `vector< vector<double> >`)

(génétique) Écrire une fonction renvoyant la position d'un codon de start dans une chaine d'ARN. Faites de même pour déterminer la position d'un codon de stop à partir d'une position donnée.

(maths) On considère les suites réelles (u_n) et (v_n) définies par leurs termes initiaux $u > 0$ et $v > 0$ et les relations de récurrence suivantes :

$$u_{n+1} = \frac{u_n + v_n}{2} \quad \text{et} \quad v_{n+1} = \sqrt{u_n v_n}.$$

On peut montrer que ces suites convergent et ont même limite, on veut vérifier expérimentalement ce fait. Écrire une fonction
`int rang (double epsilon);`
qui détermine le premier entier n tel que $|u_n - v_n| < epsilon$ (on néglige ici les erreurs d'arrondi faites par la machine).

14 Calcul numérique

14.1 Les réels.

On les représente avec les types `float`, `double` ou `long double`. Les fonctions usuelles (trigonométriques, exponentielle, logarithme, etc.) sont définies dans le fichier en-tête :
`#include <cmath>`

Rappelons que `a` à la puissance `b` se note souvent `a^b` mais se code en C++ `pow(a,b)`. Les autres différences par rapport à la notation mathématique standard sont :

- `log` : désigne le logarithme nepérien (noté ln ou Log en mathématique)
- `log10` : désigne le logarithme en base 10 (noté log en mathématique)
- `atan2(x,y)` : désigne l'argument du nombre complexe $x + iy$.
- `floor` : désigne la partie entière (attention le type C renvoyé n'est pas `int` mais reste `double` ou `float`)
- `ceil` : plus petit entier supérieur ou égal

14.2 Les nombres complexes.

Il faut inclure :
`#include <complex>`

Ensuite on choisit quel type on utilise pour la partie réelle et imaginaire, par exemple :
`complex<double> c;`
On peut définir un type synonyme pour raccourcir :
```
typedef complex<float> complexe;
complexe c1(1.2,2.3);
```
La ligne ci-dessus définit alors une variable complexe `c1` de partie réelle `1.2` et de partie imaginaire `2.3`. Il est agréable de définir un complexe `i` (en mathématique, mais en électricité on préferera définir un complexe `j`) :

`complexe i(0.0,1.0);`
pour pouvoir utiliser la notation algébrique, par exemple :
`c1=c1*(1+3*i);`
mais faites alors attention à ne plus utiliser `i` pour autre chose, par exemple comme variable de boucle !

Les fonctions usuelles sont définies (voir les réels), plus quelques fonctions propres aux nombres complexes :

- `real`, `imag`, `conj` : partie réelle, imaginaire, conjugué d'un complexe
- `polar(r,t)` : renvoie le complexe re^{it}
- `abs` : renvoie le module d'un nombre complexe (c'est un type réel).
- `norm` : renvoie le module au carré
- `arg` : argument d'un nombre complexe

14.3 Entiers en précision arbitraire et calcul formel.

Ils ne font pas partie du standard C++, il est donc nécessaire d'utiliser des librairies (et éventuellement de les installer). On utilisera la librairie C GMP que l'on peut télécharger par exemple depuis `http://gmplib.org/`. GMP version 4 possède une interface C++, rajoutez au début de votre fichier source :
`#include <gmp/gmpxx.h>`
puis remplacez le type `int` par le type `gmpxx` et rajoutez `-lgmp` à votre ligne de compilation.

Pour faire du calcul formel en C++ :
`http://www-fourier.ujf-grenoble.fr/~parisse/giac_fr.html`

Un exemple de programme avec giac :

```
// -*- mode:C++ ; compile-command: "g++ -g essai.cc -lm -lgiac -lgmp -lpng" -*-

#include <iostream>
#include <string>
#include <giac/giac.h>

using namespace std;
using namespace giac;

int main(){
  context ct;
  gen a,b("x^2-1",&ct);
  cin >> a;
  cout << _factor(a*b,&ct) << endl;
}
```

15 D'autres librairies C(++) scientifiques.

Il existe de nombreuses librairies pour éviter de reprogrammer ce qui a déjà été programmé, par exemple la Gnu Scientific Library `http://www.gnu.org/software/gsl/`.

16 Mettre au point un programme.

Lorsqu'un programme ne fonctionne pas comme prévu, on peut souvent détecter des erreurs en l'exécutant en pas-à-pas (une ligne de code après l'autre) et en examinant l'évolution du contenu des variables. L'application qui permet de faire cela s'appelle un déboggueur. Plusieurs interfaces existent, on présente ici `gdb` sous `emacs` qui s'exécute dans l'environnement d'édition du texte source (ce qui permet de corriger votre programme et de relancer le debuggueur sur la version modifiée avec les mêmes arguments et points d'arrêt).

1. Vérifiez que l'option `-g` apparait dans la ligne de compilation (première ligne du texte source par exemple :
   `// -*- mode:C++; compile-command:"g++ -g essai.cc -lm" -*-)`
2. Compilez le programme, on suppose dans la suite que le programme compilé s'appelle `a.out` ce qui est le cas par défaut
3. Dans `emacs`, sélectionnez Debugger (GDB) dans le menu Tools ou tapez sur la touche `Echap`, vous devez voir apparaitre `ESC-` au bout de quelques secondes, puis tapez `xgdb` et la touche `Entree`. Vous devez alors voir dans la ligne d'état en bas :
   `Run gdb (like this): gdb a.out`
   Modifiez si nécessaire puis tapez `Entree`. Le deboggueur est activé.
4. Pour visualiser à la fois le source du programme et le débbugueur, il peut être nécessaire de couper la fenêtre en deux (menu File->Split Window ou raccourci clavier Ctrl-X 2), et dans l'une des parties de sélectionner le source (menu Buffers ou raccourci Ctrl-X o).
5. L'étape suivante consiste à mettre un *point d'arrêt* près de l'endroit où vous suspectez qu'il y a une erreur (par exemple au début d'une fonction suspecte). Il y a deux méthodes, soit en donnant le nom de la routine, par exemple pour le programme principal `main` :
   `b main`
   soit dans le code source, en cliquant à gauche de la ligne ou en tapant `Ctrl-X` puis la touche Espace à la ligne souhaitée. On peut aussi taper `b` suivi par le numéro de ligne dans le fichier source (ou `nom_ficher:numero_ligne`). Vous pouvez mettre plusieurs points d'arrêt.
6. Puis on lance l'exécution du programme en cliquant sur Go ou en tapant `r` (pour run). On peut aussi spécifier les arguments de la fonction `main(int argc,char ** argv)` après la commande `r`.
7. Le programme s'exécute normalement jusqu'à ce qu'il atteigne le point d'arrêt. On peut alors visualiser le contenu d'une variable en utilisant l'instruction `p` (pour print), par exemple :
   `p modulo` puis touche `Entree`
   imprime le contenu de la variable `modulo` (si elle existe).
8. L'instruction `n` (next) exécute la ligne courante. Pour exécuter ensuite plusieurs lignes à la suite, on tape plusieurs fois sur la touche Entree. On peut aussi indiquer un paramètre numérique à la commande `n`.

9. L'instruction `s` (step in) permet de rentrer à l'intérieur d'une fonction si la ligne courante doit exécuter cette fonction (alors que `n` exécute cette fonction en un seul pas, sauf si cette fonction contient un point d'arrêt)
10. L'instruction `u` (until) exécute le programme sans arrêt jusqu'à ce que la ligne de code source suivante soit atteinte (ou une adresse donné en paramètre). C'est particulièrement intéressant pour sauter au-delà d'une boucle.
11. L'instruction `c` (continue) permet de continuer l'exécution jusqu'au prochain point d'arrêt. On peut aussi indiquer un paramètre pour ignorer le point d'arrêt un certain nombre de fois.
12. `kill` permet de stopper le programme en cours,
13. `d` permet de détruire un point d'arrêt (on donne alors son numéro) ou tous les points d'arrêt. On peut aussi temporairement désactiver ou réactiver un point d'arrêt (commandes `disable` et `enable`)
14. `f 0`, `f 1`, etc. permet de visualiser la fonction appelante à l'ordre demandé de la fonction interrompue (0 pour la fonction actuelle, 1 pour la 1ère appelante, etc.), ainsi que les variables de cette fonction.
15. `bt` affiche la liste des fonctions appelantes. Très utile si votre programme plante avec un `Segmentation fault`, dans ce cas, lancez le programme depuis le debuggueur sans mettre de points d'arrêt et examinez le rapport de `bt` pour voir la fonction fautive, remontez dans les appels avec `f` et visualisez la valeur des paramètres au moment de l'erreur de segmentation.
16. Vous pouvez corriger une erreur dans la fenêtre d'édition, recompiler le programme et revenir dans la fenêtre `gdb` (c'est le buffer nommé `*gud-a.out*`) puis reprenez ce mode d'emploi à l'étape 5 (relancer l'exécution du programme en tapant `r`, etc.).
17. Pour quitter le débogueur, tapez `q` (quit) ou fermez le buffer `*gud-a.out*` (activez ce buffer et choisissez `Close` dans le menu `Files`)

Remarques :

– Les variables de type structuré ou les classes s'affichent comme des structurs C, même si vous avez redéfini l'opérateur <<. Pour afficher une variable avec <<, il est judicieux de prévoir une méthode :
```
void dbgprint() const { cout << *this << endl; }
```
On peut alors visualiser une variable `x` du type considéré en tapant
```
print x.dbgprint()
```
Pour automatiser tout cela, créez un fichier nommé `.gdbinit` et contenant
```
echo Defining v as print command for class types\n
define v
print ($arg0).dbgprint()
end
```
Dans la session `gdb`, tapez `v x` pour visualiser le contenu de la variable `x`.

– Parmi les autres possibilités intéressantes de gdb, il y a la modification du contenu d'une variable en cours d'exécution (commande `print variable=valeur`) On peut aussi interrompre le programme seulement si une condition est vérifiée (`break .. if ..`).

– Pour plus d'informations, vous pouvez taper `help` dans la fenêtre du debuggueur pour obtenir l'aide en ligne ou à tout moment dans `emacs`, en tapant `Ctrl-H I`, puis `Ctrl-S gdb` puis clic de la souris avec le bouton du milieu sur `gdb`.

17 Quelques conseils

Pour obtenir des programmes lisibles il faut essayer (même lors de la création de programmes assez brefs) de repecter les règles suivantes :

– Ne pas écrire plus d'une instruction par ligne
– Utiliser des noms de variables, fonctions ... qui ont un sens
– Indenter correctement les lignes (l'indentation est automatique sous "emacs"avec la touche de tabulation)
– Laisser (de façon non aléatoire si possible) des lignes blanches
– Mettre les accolades fermantes sur une ligne à part.
– Commenter les programmes
– Lors de la conception d'un programme, dans la mesure du possible, on essaiera d'estimer le coût de l'algorithme utilisé. Un algorithme inefficace peut en effet rendre un programme inutilisable. Sur les systèmes GNU/Linux, il est possible de "profiler" les programmes de façon à étudier l'utilisation du temps machine lors de l'exécution du programme. Pour ce faire il faut compiler en utilisant l'option `-pg` (voir `man gprof`).
– Eventuellement utiliser la modularité du langage C pour créer et compiler séparément des modules ('sous-programmes´) de taille convenable (on peut automatiser la compilation en utilisant `make`). Lire la documentation de automake, autoconf, autoheader ou utilisez un logiciel de développement comme `kdevelop` pour créer un projet.

18 TP 6

(statistiques à une variables) Déterminer la moyenne, l'écart-type, la médiane, le premier et troisième quartile d'une série statistique. On pourra utiliser la fonction de tri `sort(v.begin(),v.end());` pour trier le vecteur de données.

(génétique, exercice fastidieux !) Écrire une fonction convertissant 3 codes de bases en un code de protéine. Renvoyer le codage de la protéine correspondante à une chaine d'ARN.

	U			C			A			G			
U	UUU	Phe	[F]	UCU	Ser	[S]	UAU	Tyr	[Y]	UGU	Cys	[C]	U
	UUC	Phe	[F]	UCC	Ser	[S]	UAC	Tyr	[Y]	UGC	Cys	[C]	C
	UUA	Leu	[L]	UCA	Ser	[S]	UAA	*STOP*		UGA	*STOP*		A
	UUG	Leu	[L]	UCG	Ser	[S]	UAG	*STOP*		UGG	Trp	[W]	G
C	CUU	Leu	[L]	CCU	Pro	[P]	CAU	His	[H]	CGU	Arg	[R]	U
	CUC	Leu	[L]	CCC	Pro	[P]	CAC	His	[H]	CGC	Arg	[R]	C
	CUA	Leu	[L]	CCA	Pro	[P]	CAA	Gln	[Q]	CGA	Arg	[R]	A
	CUG	Leu	[L]	CCG	Pro	[P]	CAG	Gln	[Q]	CGG	Arg	[R]	G
A	AUU	Ile	[I]	ACU	Thr	[T]	AAU	Asn	[N]	AGU	Ser	[S]	U
	AUC	Ile	[I]	ACC	Thr	[T]	AAC	Asn	[N]	AGC	Ser	[S]	C
	AUA	Ile	[I]	ACA	Thr	[T]	AAA	Lys	[K]	AGA	Arg	[R]	A
	AUG	Met	[M]	ACG	Thr	[T]	AAG	Lys	[K]	AGG	Arg	[R]	G
G	GUU	Val	[V]	GCU	Ala	[A]	GAU	Asp	[D]	GGU	Gly	[G]	U
	GUC	Val	[V]	GCC	Ala	[A]	GAC	Asp	[D]	GGC	Gly	[G]	C
	GUA	Val	[V]	GCA	Ala	[A]	GAA	Glu	[E]	GGA	Gly	[G]	A
	GUG	Val	[V]	GCG	Ala	[A]	GAG	Glu	[E]	GGG	Gly	[G]	G

(génétique) On se donne un tableau de 16 réels indicé horizontalement et verticalement par les lettres de base A, C, G, T. Chaque réel correspond à un "poids" que l'on donne si les lettres correspondantes se trouvent sur 2 chaines d'ADN, on mettra par exemple un poids positif si les lettres sont égales et négatif sinon. Écrire une fonction calculant le poids total obtenu en sommant tous ces poids pour les caractères de deux chaines de même longueur. Ceci servira ensuite à comparer une chaine à des chaines connues en trouvant la plus proche.

(maths-génétique +) Le thème est mathématique mais la méthode de programmation est extrèmement utilisée en séquençage. Le nombre de combinaisons de p éléments parmi n est noté $\binom{n}{p}$. On rappelle les relations :

$$\binom{n+1}{p+1} = \binom{n}{p} + \binom{n}{p+1}, \quad \binom{n}{0} = 1, \binom{n}{n} = 1$$

Ceci donne un algorithme de calcul de $\binom{n}{p}$ appelé triangle de Pascal. Les $\binom{n}{p}$ seront écrits à la n-ième ligne, p-ième colonne d'un tableau. On commence par écrire la 1ère ligne qui ne contient que 1. Puis chaque ligne s'obtient de la façon suivante :

- on commence la ligne par un 1 qui correspond à $p = 0$
- ensuite on obtient une valeur en additionnant deux valeurs de la ligne précédente : celle de la même colonne située juste au-dessus et celle de la colonne précédente située juste au-dessus
- On termine par un 1 qui correspond à $p = n$.

Par exemple, on obtient les lignes suivantes :

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Ecrire un programme mettant en oeuvre cet algorithme.