Compilation
Eléments de syntaxe

Code source et exécutable

Le langage C est un langage informatique qui n'est pas compréhensible directement par l'ordinateur. Celui ci ne connaît que le langage machine, qui est une suite de 0 et de 1. Il faut donc qu'une traduction soit effectuée entre le programme en langage C, que l'on appelle le **code source**, et le résultat en langage machine, qui est **exécutable** par l'ordinateur. les exécutables sont, par exemple avec le système Windows, les fichiers dont l'extension est *.exe*.

Le code source est contenu dans un fichier avec l'extension **.c**, l'exécutable est contenu dans un fichier avec l'extension **.exe** sous Windows, ou avec l'attribut '**x**' (exécutable) sous Unix/Linux, il apparaît dans ce cas avec une ***** derrière le nom de fichier.

La traduction

Cette traduction est effectuée par un programme (en fait par plusieurs programmes à la suite) qui analysent le code source pour voir s'il est correct, au niveau de la syntaxe, et le traduisent s'il n'y a pas d'erreurs. Elle se fait en fait en 2 phases, à partir du fichier **.c**.

Compilation

La compilation est la première phase de la traduction, et cette étape génère un fichier dans un langage intermédiaire entre le langage C et le langage machine. Ce fichier porte en général le même nom que le fichier source, mais avec l'extension .o au lieu de l'extension .c. Ce fichier n'est pas éditable (il ne contient rien de compréhensible), mais n'est pas non plus un exécutable.

Edition des liens

L'édition des liens permet de créer le fichier exécutable à partir du fichier .o. Cette étape a pour rôle de faire en sorte que le système d'exploitation puisse lancer le programme, ce qui nécessite d'ajouter d'autres informations que celles contenues dans le programme.

Dans la pratique

Environnement de développement

Le processus de compilation + édition des liens peut être réalisé automatiquement par certains environnements de développement (comme Dev-C++ ou Visual C++), mais on peut aussi le réaliser pas à pas, ce que nous allons faire pour bien voir la manière dont se déroulent ces phases. Nous utiliserons donc une fenêtre DOS par laquelle nous ferons la compilation

puis l'édition des liens. Cette technique s'apparente à celle que l'on utilise avec les systèmes Unix / Linux.

Commandes pour compiler

Le compilateur utilisé est un programme nommé **g++** (***g**nu **c**++ **c**ompiler*), qui est capable de faire les 2 étapes de traduction.

Pour faire la compilation, il faut utiliser le programme nommé g++.exe, dont le nom complet est E:\apps\Dev-Cpp\bin\g++.exe. Pour éviter d'avoir à taper entièrement ce nom, nous allons mettre à jour la variable d'environnement nommé `path` qui contient une liste de répertoires à explorer pour trouver un programme. Pour cela, tapez la commande suivante :

```
path=%path%;E:\apps\Dev-Cpp\bin
```

pour savoir si cette mise à jour a bien été effectuée, tapez la commande

```
g++
```

vous devriez obtenir le message suivant :

```
g++: no input files
```

Si vous obtenez celui-ci :

```
'g++' is not recognized as an internal or external command,
operable program or batch file.
```

c'est que la mise à jour ne s'est pas bien faite, vérifiez bien que vous n'avez pas fait de faute de frappe !

nous pouvons maintenant passer à la compilation proprement dite :

rappel : pour compiler un fichier .c, on utilise la commande suivante :

```
g++ -c nom_du_fichier.c
```

donc, pour notre exemple, on utilisera la commande :

```
g++ -c hello.c
```

le compilateur indique s'il trouve des erreurs et éventuellement peut indiquer des avertissement (warning), qui n'empêchent pas la compilation mais peuvent gêner l'exécution du programme.

Vérifiez que vous avez maintenant dans le répertoire TP1 le fichier hello.o. S'il ne s'y trouve pas, le compilateur a du vous signaler au moins 1 erreur qui a rendu la compilation impossible.

Edition des liens

Dernière étape avant l'obtention du fichier exécutable : l'édition des liens. A partir du fichier .o obtenu par compilation, on génère le fichier exécutable à l'aide de la commande suivante :

```
g++ nom_du_fichier.o -o nom_du_fichier.exe
```

dans notre exemple, cette commande est donc :

```
g++ hello.o -o hello.exe
```

on peut choisir de nommer le fichier exécutable autrement, à la seule condition que son extension soit .exe.

exemple :

```
g++ hello.o -o bonjour.exe
```

Après cette commande, vous devez normalement avoir un fichier hello.exe (ou d'un autre nom si vous avez choisi un autre nom) dans votre répertoire.

Les particularités du langage C

Eléments de syntaxe

Pour faire un programme en C, il ne suffit pas de faire la traduction depuis le langage algorithmique, même si on peut former de cette manière 90 % du code source. Il faut ajouter quelques éléments supplémentaires : pour l'instant, nous écrirons tous nos programmes de la manière suivante :

```
#include <stdlib>
#include <iostream>

using namespace std;

int main(int agrc, char *argv[])
{
      // définition des variables
      // instructions du programme

      system("pause");
      return 0;
}
```

A quoi correspondent ces lignes et instructions ?

Nous mettrons volontairement de côté les lignes :

#include <cstdlib> et **using namespace std;**

Elles sont indispensables, mais nous n'avons pas besoin de savoir pourquoi pour le moment. Rappelez-vous qu'elles doivent être présentes dans le programme.

```
#include <iostream>
```

Ce n'est pas vraiment une instruction, mais une directive, qui permet d'utiliser les instructions d'affichage et de saisie.

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

cette ligne correspond au mot programme que l'on utilise en langage algorithmique. Elle indique le début du programme.

```
{
```

les instructions du programme sont, comme pour les blocs d'instruction, entourés par des accolades '{' et '}'.

```
// définition des variables
// instructions du programme
```

on place ensuite les définitions de variables du programme et les instructions, nous en verrons la syntaxe au fur et à mesure de leur utilisation.

```
system("pause");
```

Cette instruction permet d'insérer une pause entre la fin du programme et la fermeture de la fenêtre du programme. Si on l'omet, la fenêtre se ferme tout de suite si on exécute directement le fichier .exe à partir de Windows. Un message "`press any key to continue . . .`" apparaît, il suffit d'appuyer sur une touche pour fermer la fenêtre.

```
return 0;
```

cette instruction est indispensable pour terminer le programme, nous reviendrons plus tard sur sa signification.

```
}
```

cette accolade correspond à l'accolade ouvrante du début du programme.

Le programme "hello world"

La première instruction que nous allons traiter est l'instruction qui correspond à l'affichage.

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
	cout << "hello world !" << endl; // instruction d'affichage

	system("pause");
	return 0;
}
```

Si l'on exécute ce programme, le message apparaît à l'écran :

```
hello world
press any key to continue . . .
```

Définition des variables en langage C :

En langage C, les variables sont définies de la même manière qu'en langage algorithmique : on donne leur type, suivi du nom de la variable et d'un point virgule ';'.

Les noms des types en langage C :

Les entiers

Le nom du type pour les entiers est **int**, cependant on l'utilise rarement seul car il n'est pas défini de manière très stricte : selon les machines, l'intervalle des nombres représentables n'est pas le même. On utilise, pour bien préciser l'intervalle, ce qu'on appelle des modificateurs de type que l'on place avant le nom de type **int** :

`short` : intervalle court

`long` : intervalle long

On peut aussi préciser si l'on veut que l'intervalle ne comprenne que des nombres positifs, en ajoutant le modificateur **unsigned**, ou, si l'on veut que l'intervalle comprenne des nombres positifs et négatifs, le modificateur **signed**.

On obtient donc toutes les possibilités suivantes, résumées dans un tableau :

Nom complet du type	Nom abrégé	Intervalle
`unsigned short int`	`unsigned short`	[0; 65535]

`signed short int`	`short`	[-32768 ; +32767]
`unsigned long int`	`unsigned long`	[-2 147 483 648, +2 147 483 647]
`signed long int`	`long`	[0, +4 294 967 295]

Note : on peut employer le nom abrégé au lieu du nom complet, le langage C fera de lui-même la correspondance. Par défaut, on a tendance a employer **long** comme nom de type. On a le droit d'employer **int** comme nom de type, dans ce cas les valeurs seront positives ou négatives, mais on ne connaît pas les valeurs de l'intervalle.

Les caractères

Le nom du type pour les caractères est **char**.

Les réels

Il existe, pour les réels, deux noms de type, **float** et **double**. La distinction entre les deux types réside dans l'intervalle des valeurs représentables :

- Pour le type **float** : de -10^{38} à -10^{-38} et de 10^{-38} à 10^{38}
- Pour le type **double** : de -10^{308} à -10^{-308} et de 10^{-308} à 10^{308}

Par convention, en langage C, on utilise systématiquement le type `double` pour stocker un nombre à virgule.

On peut aussi employer les définitions multiples, avec la même syntaxe que pour le langage algorithmique.

Retour sur l'instruction cout <<

L'instruction `cout <<` correspond à l'ordre `afficher()` du langage algorithmique, mais sa syntaxe est différente.

Lorsque l'on veut afficher du texte, la syntaxe est la même : il suffit de le mettre entre doubles guillemets, comme dans l'exemple précédent.

Par contre, pour afficher des variables, la syntaxe est modifiée : on peut alterner texte et variables, en utilisant des chevrons << pour les séparer.

La liste des variables à afficher est précisée après le texte, derrière une virgule ','.

Quelques exemple : on veut afficher la valeur d'une variable a, de type entier (définie par `entier a;`), avec le texte suivant :

```
cout << la valeur " << a << " est stockée dans la variable a";
```

En langage C; la variable a est définie par :

```
long a;
```

ou

```
int a;
```

(nous traiterons les deux cas).

exemple : affichage de variables et d'expressions :

On veut traduire l'instruction suivante en langage C :

```
afficher("le produit de ", x, " et de ", y " vaut ", x*y);
```

On devra donc afficher trois valeurs de type réel dans le texte affiché à l'écran :

```
cout <<"le produit de " << x <<" et de " << y << " vaut " << x*y << endl;
```

Instruction de saisie : cin >>

Utilisation :

```
Cin >> nom_de_variable;
```

Comment traduire les instructions suivantes d'un programme en langage algorithmique :

```
entier a;
saisir(a);
```

en langage C :

```
long a;
cin >> a ;
```

du langage algorithmique :

```
reel x;
saisir(x);
```

en langage C :

```
double x;
cin >> x;
```

du langage algorithmique :

```
caractere ch;
saisir(ch);
```

en langage C:

```
char ch;
cin >> ch;
```

Opérateurs de base et expressions

Affectation

L'affectation en langage C est réalisée par l'opérateur :

=

Ce qui est un choix contestable, car ce symbole est naturellement compris comme étant le symbole d'égalité. Le problème vient du fait que le symbole ← n'est pas employé par le langage C. Cela introduit malheureusement une confusion avec le symbole d'égalité, qui a donc également une syntaxe différente en langage C.

Exemple de traduction :

```
programme eq_degre_2

reel a, b, c, delta;

afficher("entrez les valeurs de
a, b et c:");

saisir(a);

saisir(b);

saisir(c);

delta ← b*b-4.0*a*c;
```

Langage algorithmique

```
#include <cstdlib>
#include <iostream>

int main()
{
double a, b, c, delta;

cout <<"entrez les valeurs de
a, b et c:";
cin >> a;
cin >> b;
cin >> c;

delta = b*b-4.0*a*c;

return 0;
}
```

Langage C

Opérateurs mathématiques

Les opérateurs mathématiques de base sont les mêmes en langage C et en algorithmique :

```
+ : addition
```

```
- : soustraction
* : multiplication
/ : division
% : modulo (reste de la division entière)
```

Leur comportement, notamment vis à vis des divisions entre nombres entiers est également le même : si l'on divise en C deux entiers, on obtient deux entiers :

Exemple de traduction :

```
programme divis_entier

entier divid ,divis, quot, rem;

divid ← 11;
divis ← 4;

quot ← divid / divis;
rem ← divid % divis;

afficher(divid , " = ", quot," x
", divis," + ",rem);
```

```
#include <iostream>

int main()
{
      long divid, divis, quot,
rem;

      divid = 11;
      divis = 4;

      quot = divid / divis;
      rem = divid % divis;

      cout << divid << " = " <<
quot << " x " << divis << " + " <<
rem << endl;

      return 0;
}
```

Opérateurs logiques

Comparaisons

Les opérateurs de comparaison sont sensiblement les mêmes que ceux du langage algorithmique, si ce n'est les opérateurs d'égalité et d'inégalité, que nous emploierons souvent. Il faut donc porter une certaine attention

Opérateur	Langage algorithmique	Langage C
Egalité	=	**==**
Inégalité	≠	**!=**
Supérieur strict	>	>
Supérieur large	>=	>=
Inférieur strict	<	<
Inférieur large	<=	<=

L'opérateur d'inégalité

Le caractère ≠ n'étant pas aisément accessible au clavier (voilà, par exemple, à quoi peut tenir la syntaxe d'un langage informatique…), plusieurs choix sont possibles, mais la seule syntaxe qu'accepte le langage C est : **!=**

Même si l'utilisation d'autres notations, telles que **<>**, sont tentantes (et, après tout, légitimes), le compilateur ne manquera pas de vous faire savoir qu'il ne les accepte pas.

L'opérateur d'égalité

Puisque le symbole = est utilisé par le langage C pour l'opérateur d'affectation, il est nécessaire d'utiliser un autre symbole, celui qui est retenu pour le C est : **==**

Le choix de cette notation est hélas assez malheureux, pour plusieurs raisons :

- Il ressemble beaucoup au symbole =, donc n'est pas d'un usage intuitif;
- Le compilateur n'est d'aucun secours pour l'emploi de ce symbole, car il comprend les deux symboles (affectation et opérateur d'égalité) et ne génère aucune erreur lorsqu'une affectation est faite au lieu d'un test d'égalité; cependant les résultats obtenus en remplaçant un symbole par l'autre peuvent être totalement différents !

Opérateurs booléens

Opérateur	Langage algorithmique	Langage C
Conjonction	ET	&&
Disjonction	OU	||
Négation	NON	!

Le symbole correspondant au OU est une double barre verticale obtenue au clavier en pressant simultanément les touches 'Alt Gr' (juste à droite de la barre d'espacement) et la touche '6' (pas celle du clavier numérique).

Structures de contrôle

Alternatives et sélection

Test simple : si…alors

En langage algorithmique :

```
si condition alors
{
      instructions;
}
```

Avec le langage C :

```
if (condition)
{
      instructions;
}
```

Notez qu'il n'y a pas d'équivalent pour la partie 'alors', certains langages (comme le Pascal) utilisent 'then' pour matérialiser le 'alors'.

Test : si…alors…sinon

En langage algorithmique :

```
si condition alors
{
      instructions1;
}
sinon
{
      instructions2;
}
```

En langage C, le 'sinon' est traduit par son homologue anglais : else, qui s'utilise de la même manière :

```
if (condition)
{
      instructions1;
}
else
{
      instructions2;
}
```

Alternatives multiples : si …alors..sinon si…alors…sinon

En langage algorithmique :

```
si condition1 alors
{
      instructions1;
}
sinon si condition2 alors
{
      instructions2;
}
…
sinon
{
      instructionsN;
}
```

En langage C, la traduction est immédiate en reprenant les éléments vus précédemment : il s'agit presque d'une traduction littérale.

```
if (condition1)
{
      instructions1;
}
else if (condition2)
{
      instructions2;
}
…
else
{
      instructionsN;
}
```

rappel : ce type de structure est particulièrement adapté aux conditions exclusives, ou lorsque l'on veut être certain de traiter tous les cas possibles.

Affecter au lieu de tester

Il s'agit malheureusement d'une erreur classique, difficilement repérable (même pour des yeux bien exercés), et au résultat parfois surprenants.

Valeur d'une affectation : En langage C, une affectation est aussi une expression, donc a une valeur…et cette valeur peut être comprise comme VRAI ou FAUX par le compilateur, qui ne trouve donc rien à redire à ce genre d'écriture : (on cherche à écrire un programme qui indique si une variable est nulle ou non).

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
      long c;

      c = 5;

      if (c=1)
      {
            cout << "la variable c vaut 1 !" << endl;
      }

      return 0;
}
```

Alors que le test d'égalité aurait du se faire de la manière suivante :

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
      long c;

      c = 5;

      if (c==1)
      {
            cout << "la variable c vaut 1 !" << endl;
      }

      return 0;
}
```

Que se passe-t-il avec la première version du programme, que le compilateur accepte parfaitement ? A la place de c==1 (test), on trouve (c=1) : affectation. Donc, dans un premier temps, la variable c reçoit la valeur 1 (ce qui n'est pas le but recherché). De plus, une affectation est considérée par le langage C comme une expression dont la valeur est égale à la valeur rangée dans la variable. Dans notre exemple, c=1 est une expression qui vaut…1 !

Le langage C, comme le langage algorithmique, fait une association entre les entiers et les valeurs VRAI et FAUX. Ainsi, la valeur 1 est comprise comme étant VRAI, et l'instruction d'affichage, concernée par la structure if, est donc effectuée !

Un autre exemple :

Que fait le programme suivant ?

```
#include <stdio.h>

int main()
{
      long c;

      c = 5;

      if (c=0)
      {
            cout << "la variable c vaut 0 !" << endl;
      }
      else
      {
            cout << "c n'est pas nul : c = " << c << endl;
      }
```

```
      return 0;
}
```

selon…cas

Cette structure est un peu plus délicate à utiliser en langage C qu'elle ne l'est en langage algorithmique

En langage algorithmique :

```
selon (variable ou expression entière)
{
      cas valeur1 : bloc d'instructions 1;
      cas valeur2 : bloc d'instructions 2;
      …
      cas valeurn : bloc d'instruction n;
      par defaut : bloc d'instruction n+1;
}
```

En langage C, on ne peut se contenter d'une traduction littérale, car le comportement du programme n'est pas celui auquel on pourrait s'attendre : il faut ajouter, à la suite de chaque série d'instructions correspondant à une valeur l'instruction `break`, dont nous allons détailler le fonctionnement.

`selon` est traduit par `switch`, `cas` est traduit par `case`.

```
switch(expression_entiere)
{
      case valeur1 :  instructions1;
                      break;
      case valeur2:   instructions2;
                      break;
      case valeurn :  instructionsn;
                      break;
      default :       instructions_par_defaut;
                      break;
}
```

En langage C, l'instruction break permet de terminer la boucle ou la structure en cours, et de continuer le programme aux instructions qui suivent directement cette structure. On ne l'utilise normalement jamais, sauf dans le cas précis de la structure switch…case, car cette dernière a un comportement particulier :

Lorsque le langage C trouve un cas (`case`) correspondant à la valeur ou à l'expression entière concernée par le `switch`, il fait le bloc d'instruction correspondant (ce que l'on attend), mais aussi **tous les blocs suivants**, jusqu'au dernier (le bloc correspondant à

`default`). Il faut donc, dès qu'un cas a été reconnu, terminer la structure `switch...case` après que le bloc d'instructions correspondant a été effectué : cela est réalisé en faisant suivre systématiquement ce bloc de l'instruction `break`.

Boucles, répétitions

Boucle tant que

En langage algorithmique :

```
tant que (condition)
{
      instructions;
}
```

en langage C : il s'agit d'une traduction littérale :

```
while (condition)
{
      instructions;
}
```

Boucle faire...tant que

En langage algorithmique

```
faire
{
      instructions;
}tant que  (condition);
```

en langage C : encore une traduction littérale (ce qui est confortable)

```
do
{
      instructions;
} while(condition);
```

Boucle pour

En langage algorithmique :

```
pour variable de A à B faire
{
      instructions;
}
```

en langage C :

```
for(variable=A; variable <= B; variable = variable+1)
{
      instructions;
}
```

la boucle for en langage C est beaucoup plus souple que la boucle pour de l'algorithmique car son fonctionnement est en fait le suivant : la boucle for comporte 3 rubriques :

- initialisations;
- conditions;
- incrémentations;

Et est écrite de la manière suivante :

```
for(initialisations; conditions; incrémentations)
{
      instructions;
}
```

Ces rubriques peuvent, a priori, comporter n'importe quelle instruction.

Le déroulement de la répétition avec `for` est le suivant :

1) l'instruction (ou les instructions) de la rubrique initialisations est (sont) faite(s).
2) la condition (rubrique 2) est testée :
   - si elle est VRAI, alors les instructions du bloc sont faites, puis la (les) instruction(s) d'incrémentation (rubrique 3) est également faite, et il y a un retour à l'étape 2) : test de la condition. On ne recommence pas les instructions de la rubrique 1.
   - Si elle est FAUX, la boucle se termine et l'on passe aux instructions suivantes.

Dans ce sens, la boucle `for` est un équivalent de la boucle `while` (car la condition est testée avant d'effectuer le bloc d'instructions).

Illustration avec l'exemple fourni :

```
for(variable=A; variable <= B; variable = variable+1)
{
      instructions;
}
```

rubrique 1 : initialisation : l'instruction `variable=A` est faite une fois (et ne sera pas refaite)

rubrique 2 : conditions : `variable <= B` : cette condition sera testée pour savoir si les instructions sont faites ou non

rubrique 3 : incrémentation : `variable = variable + 1` : cette instruction sera faite après les instructions du bloc et avant de refaire le test de la condition : c'est exactement comme si cette instruction était la dernière du bloc d'instructions.

Cette boucle `for` est strictement équivalente à :

```
variable = A;

while (variable <= B)
{
      instructions;
      variable = variable +1 ;
}
```

les instructions de rupture de séquence

Vous serez peut être amenés à rencontrer les instructions suivantes dans un ouvrage ou sur un site Internet traitant du langage C; or l'utilisation de ces instructions est en général déconseillé (voire même interdit) lorsque l'on fait de la programmation structurée, car le rôle de ces instructions est justement de modifier le déroulement des structures de contrôle du programme.

Tous les programmes et problèmes peuvent être traités sans ces instructions. S'il apparaît dans un de vos programmes que vous ayez besoin d'une de ces instructions, c'est que sa structure est mal conçue à l'origine, et il vaut mieux dans ce cas la reprendre.

l'infâme goto

Cette instruction permet de se rendre directement à un endroit du programme repéré par un nom ou label : elle rend les programmes illisibles et très durs à comprendre. Les dangers de l'utilisation de cette instruction sont connus depuis maintenant plus de 30 ans. Cette instruction est **définitivement** à bannir !

l'instruction continue : un fossile de l'informatique

L'instruction `continue`, utilisée avec les boucles `while`, `do...while` et `for`, permet de terminer l'itération en cours et de revenir au test de la condition de boucle : elle est facilement remplaçable par un simple test `if...`, et donc ne doit **jamais** apparaître dans un programme structuré.

l'instruction break

L'instruction `break`, déjà évoquée dans la gestion de la sélection avec `switch...case` termine l'exécution de la structure en cours. A l'exception de `switch...case`, cette instruction ne doit également **jamais** apparaître dans un programme structuré.

Bien écrire un programme en langage C

Dans cette partie, nous ne nous intéresserons qu'à la mise en forme du programme.

Le choix des noms de variables

Ce thème a déjà été abordé en algorithmique lors des tous premiers cours : tout ce qui a été dit reste valable pour le langage C : choisissez des noms respectant les règles de syntaxe, bien évidemment, et qui soient des noms pertinents et pratiques à utiliser.

Les commentaires

L'introduction de commentaires dans un programme en langage C suit exactement la même syntaxe que pour le langage algorithmique. Un commentaire est un texte qui est ignoré par le compilateur, donc ne fait pas partie à proprement parler du programme, mais qui permet de fournir au lecteur des explications sur le fonctionnement du programme : il ne faut pas hésiter à en placer, sans toutefois noyer le programme !

Rappel sur la syntaxe des commentaires :

// (double barre de division) placé sur une ligne débute un commentaire qui se termine à la fin de la ligne correspondante. Si l'on désire faire tenir un commentaire sur plusieurs lignes, il faut mettre le symbole // au début de chaque ligne de commentaire

/* début un commentaire qui peut tenir sur plusieurs lignes. Un commentaire débuté par /* doit être terminé par */

exemples :

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
      long a,b,c; // commentaire sur une ligne

      cout << "entrez la valeur de a:";

      /* commentaire sur plusieurs lignes
      on ne rappelle pas le début de commentaire à chaque ligne
      on devra par contre écrire le symbole de fin de commentaire
```

```
      */
      cin >> a; // saisie

      // autres instructions et fin de programme

      return 0;
      /* prog termine */
}
```

L'indentation

- L'indentation d'un programme fait partie de certaines règles de mise en page afin de rendre ce programme le plus clair et lisible possible. La mise en page et la présentation d'un programme ne changent absolument rien à la manière dont il se déroule, mais peut grandement modifier la facilité avec lequel on le comprend. L'indentation concerne la mise en forme des blocs d'instructions délimités par les accolades { et }. Ces accolades sont utilisées : pour délimiter les instructions du programme;
- Pour délimiter les blocs correspondants aux structures de contrôle.

Dans les deux cas, la règle d'indentation est la suivante :

- Une accolade ouvrante se situe au même niveau que l'instruction qui la précède;
- Une accolade fermante se situe au même niveau que l'accolade qui ouvre le bloc fermé par l'accolade fermante;
- Une instruction suivant une accolade fermante se trouve au même niveau que cette accolade;
- Une instruction suivant une accolade ouvrante est décalée vers la droite par rapport à cette instruction;

Illustration : deux versions d'un programme

Sans indentation, sans commentaires, avec des noms de variables les plus courts possibles.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(){
long a,b;
      double x,y;
cout << "saisissez x:");cin>>x;
```

```
      cout<<"entrez n:";
cin>>a;y=1.;
b=0; while(b < a) {y=y*x;
b=b+1;
} cout <<"y="<<y;
      }
```

Avec indentations, commentaires et choix de noms de variables pertinents.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
      long n,cpt;
      double x,puiss;

      // saisie des valeurs de x et n pour calculer x^n

      cout <<"saisissez x:";
      cin >> x;

      cout << "entrez n:";
      cin >> n;

      // initalisation des autres variables et calcul

      cpt=0; // valeur initiale du compteur
      puiss = 1.0; // initialisation, x^0 = 1

      while(cpt < n) // calcul de x^n par accumulation de produits
      {
            puiss = puiss*x;
            cpt = cpt+1;
      }

      // affichage du résultat

      cout <<"x^n="<<puiss;

      return 0;
}
```

A votre avis, quelle est la version que préfèrera un lecteur ou un correcteur ?