Le Langage C++

Patrick TRAU - ULP IPST version du 02/10/05

1) Introduction

1.1) Organisation - fonctionnement de l'ordinateur

Je suppose que vous savez utiliser un ordinateur, et comprenez comment il fonctionne. Voir cours d'IUP1 (http://www-ipst.u-strasbg.fr/pat/internet/techinfo/).

1.2) Langages de programmation

Un ordinateur est une machine bête, ne sachant qu'obéir, et à très peu de choses :

- addition, soustraction, multiplication en binaire, uniquement sur des entiers,
- sortir un résultat ou lire une valeur binaire (dans une mémoire par exemple),
- comparer des nombres.

Sa puissance vient du fait qu'il peut être **programmé**, c'est à dire que l'on peut lui donner, à l'avance, la séquence (la suite ordonnée) des ordres à effectuer l'un après l'autre. Ces ordres, codés en binaire, sont sauvés dans un fichier nommé « exécutable » (.exe sous Windows). Le grand avantage de l'ordinateur est sa rapidité. Par contre, c'est le programmeur qui doit TOUT faire.

L'ordinateur ne comprenant que des ordres codés en binaire (le langage machine) peu pratiques d'emploi, des langages dits "évolués" ont été mis au point pour faciliter la programmation, au début des années 60, en particulier FORTRAN (FORmula TRANslator) pour le calcul scientifique et COBOL pour les applications de gestion. Puis, pour des besoins pédagogiques principalement, ont été créés le BASIC, pour une approche simple de la programmation, et PASCAL au début des années 70. Ce dernier (comme le C) favorise une approche méthodique et disciplinée (on dit "structurée").

Le C a été développé conjointement au système d'exploitation UNIX, dans les Laboratoires BELL, par Brian W Kernigham et Dennis M Ritchie, qui ont défini en au cours des années 70, dans "The C Language", les règles de base de ce langage. Le but principal était de combiner une approche structurée (et donc une programmation facile) avec des possibilités proches de celles de l'assembleur (donc une efficacité maximale en exécution, quitte à passer plus de temps de programmation), tout en restant standard (c'est à dire pouvoir être implanté sur n'importe quelle machine). Puis ce langage a été normalisé (norme ANSI), cette norme apportant un nombre non négligeable de modifications au langage. Contrairement au Pascal, ce langage est principalement destiné aux programmeurs confirmés, il sera donc avare en commentaires et vérifications, supposant que le programmeur sait ce qu'il fait.

A la fin des années 80, Bjarne Stroustrup crée le C++, qui garde toutes les possibilités du C, l'améliorant (commentaires, constantes, passage d'arguments par adresse, arguments par défaut...) mais surtout en y ajoutant les objets (encapsulation, héritage, polymorphisme, surcharge...). Le C++ combine donc toutes les possibilités de la programmation «classique» et la puissance de l'approche «objets». Je préciserai quelles

fonctionnalités sont spécifiques au C++ (les autres fonctionnant également en C ANSI)

100% COURS
MCours.com
Cours et mémoires gratuits
mcourscom@gmail.com

2) Le processus de compilation

Le C++ est un langage compilé, c'est à dire qu'il faut :

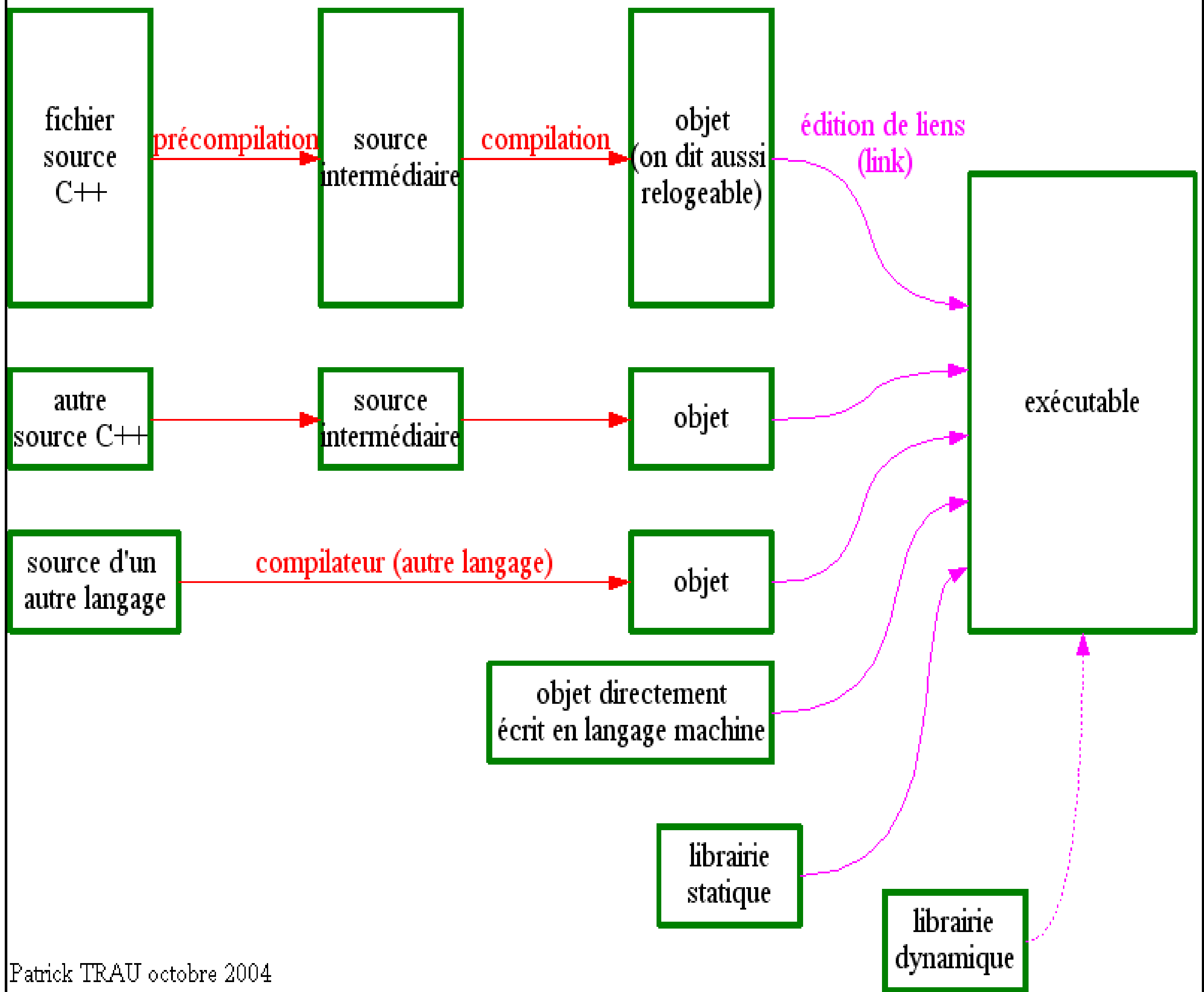
- entrer un texte dans l'ordinateur (à l'aide d'un programme appelé EDITEUR, par exemple kwrite ou le blocnotes),
- le traduire en langage machine (c'est à dire en codes binaires compréhensibles par l'ordinateur) : c'est la compilation (nous utiliserons le compilateur g++),
- l'exécuter.

Pour aller plus en détails (voir schéma) : Pour commencer, la compilation d'un source C++ se fait en plusieurs phases. La première (la précompilation) permet des modifications du code source, et ce à l'aide de « directives » (commençant par #). Ce n'est que dans une seconde phase que ce source intermédiaire est effectivement traduit en langage machine : l'objet.

Un programme exécutable peut être fabriqué à partir de divers éléments. On peut par exemple décomposer son programme source C++ en plusieurs fichiers, que l'on compile séparément. Cela permet, en cas de modification d'un seul source, de ne recompiler que celui-ci (et réutiliser les autres objets tels quels). Cela permet également d'utiliser un même objet dans différents programmes (s'ils nécessitent une fonction commune). On peut également utiliser des objets générés à partir d'un autre langage (si, quand on les a écrits, on utilisait un autre langage, ou si quelqu'un nous a donné ses sources).

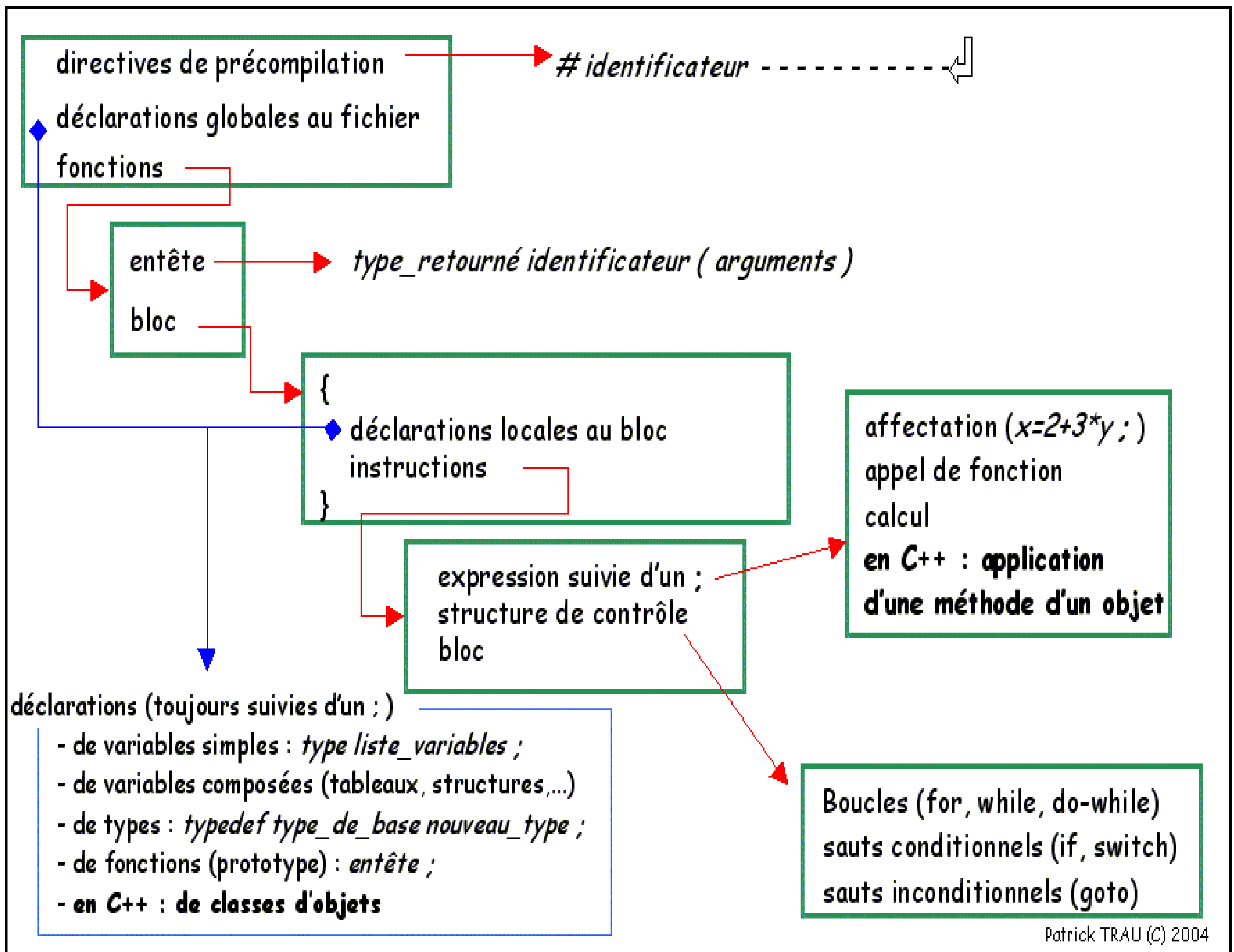
Tout ces objets doivent être regroupés en un seul programme exécutable : c'est le « link ». Il permet également de lier diverses librairies. Les librairies sont des objets qu'on nous a fournies toutes faites : fonctions d'accès aux ressources de l'ordinateur, fonctions graphiques, bibliothèques de fonctions mathématiques... A la différence d'un objet, seules les parties utiles d'une librairie seront chargées dans l'exécutable.

Pour qu'un programme puisse être exécuté, il faut qu'il soit complètement chargé dans la mémoire de l'ordinateur. C'est pourquoi tous les objets et librairies statiques sont incorporés dans l'exécutable (qui pourra donc devenir assez gros). Dans les systèmes d'exploitation multitâches, si plusieurs programmes utilisent une même librairie, elle serait alors copiée plusieurs fois en mémoire. C'est pourquoi on utilise des librairies dynamiques (.dll sous Windows) : l'exécutable sait simplement qu'elle est nécessaire. Si, lors du chargement du programme, elle est déjà présente, inutile de la charger une seconde fois. Certains systèmes ont du mal par contre à décider quand supprimer une librairie dynamique de la mémoire (en particulier si l'un des programmes l'ayant demandé à planté).



3) Structure d'un fichier source, définitions

Voici un schéma définissant cette structure. Il nous servira tout au long du cours.



Regardons ce petit programme, et tentons d'en repérer les différentes parties :

```
1)      /* premier exemple de programme C++ */
2)      #include <iostream.h>
3)      #define TVA 19.6
4)      int main(void)
5)       {
6)        float HT,TTC;    //on déclare deux variables
7)        cout<<"veuillez entrer le prix HT : ";
8)        cin>>HT;
9)        TTC=HT*(1+(TVA/100));
A)        cout<<"prix TTC : "<<TTC<<"\n";
B)       }
```

3.1) identificateurs, séparateurs, commentaires.

Définissons quelques termes importants.

Le **commentaire** est une partie du fichier source qui n'est pas prise en compte par le compilateur. En C, un commentaire commence par « /* » et finit par « */ » (voir ligne 1). Il peut y avoir tout ce qu'on veut entre (y compris plusieurs lignes), sauf un commentaire du même type (on dit qu'ils ne peuvent pas être imbriqués). En C++ on peut également utiliser le commentaire commençant par « // » et se terminant à la fin de la même ligne. Ce second type de commentaire peut par contre être imbriqué à l'intérieur d'un commentaire /* ... */.

Le **séparateur** en C/C++ sert à séparer des mots. Il peut être un espace, une tabulation, un retour à la ligne (sauf dans les directives de précompilation), un commentaire, ou une combinaison de plusieurs d'entre eux. Vous pouvez donc aller à la ligne à tout endroit où l'on peut mettre un espace, ou à l'inverse regrouper sur une même ligne plusieurs instructions séparées par un espace.

Un **identificateur** est un nom que le programmeur donne à une entité. Un identificateur est composé de lettres (majuscules ou minuscules non accentuées, une majuscule étant considérée comme différente de sa minuscule, donc de A à Z et de a à z), de chiffres (0 à 9) et du signe «souligné » (underscore). Commencez le toujours par une lettre (et utilisez généralement des minuscules). Un identificateur est terminé soit par un séparateur, soit parce qu'il est suivi d'un caractère interdit (ni lettre ni chiffre ni souligné). Dans ce dernier cas, vous avez néanmoins le droit d'insérer un séparateur entre l'indentificateur et ce caractère. Par exemple, dans la ligne 4 il FAUT mettre au moins un espace entre int et main (sinon cela forme un seul identificateur), on peut (mais n'est pas obligé) insérer un séparateur devant et derrière les parenthèses. Il est par contre impossible d'insérer un séparateur au milieu d'un identificateur.

3.2) directives du précompilateur

La première chose que fait le compilateur, c'est d'appliquer les directives de précompilation. Elles commencent toujours par #, et se finissant à la fin de la ligne (il n'y en a donc qu'une par ligne). Elles sont appliquées lors de la précompilation, indépendamment des règles du C++ (ce n'est que le résultat qui doit obligatoirement respecter les règles du compilateur). Une directive s'applique depuis l'endroit où elle est écrite jusqu'à la fin du fichier (c'est pourquoi on les place souvent au début du fichier).

`#define identificateur texte de remplacement` : à chaque fois que le précompilateur rencontrera (dans la suite du fichier) l'identificateur, il le remplacera par le texte de remplacement. Dans notre exemple (ligne 3), il remplacera le mot TVA par 19.6 partout. Ces transformations sont faites lors de la précompilation, où l'on ne fait que du "traitement de texte", c'est à dire des remplacements d'un texte par un autre sans chercher à en comprendre la signification. Attention, le premier espace (après define) définit le début de l'identificateur (sans espace); le second définit le début du texte de remplacement (qui se finit à la fin de la ligne donc on peut y mettre des espaces). On peut par exemple se servir de cela pour définir un nouveau langage francisé :

```
#define si if
#define sinon else
#define tant_que while
etc...
```

On peut également définir des macros :

Si l'on définit : `#define carre(a) a*a`
alors `carre(x)` sera transformé en `x*x` (ça convient bien), mais `1+carre(x+1)` en `1+x+1*x+1` (qui vaudra 1+x+(1*x)+1).

Alors qu'avec : `#define carre(a) ((a)*(a))`
`carre(x)` sera transformé en `((x)*(x))` (convient encore) et `1+carre(x+1)` en `1+((x+1)*(x+1))` (ce qui est bien ce que l'on attendait).

#include "nomdefichier" : inclure un fichier à la place de cette ligne. Ce fichier est défini par le programmeur, il est cherché dans le même répertoire que le fichier source. Vous pourriez par exemple mettre tous les #define nécessaires pour franciser le langage dans le fichier «monlangage.inc » et l'inclure dans tous vos programmes par `#include "monlangage.inc"`

#include <nomdefichier> : inclut un fichier système (fourni par le compilateur ou le système d'exploitation). Vous n'avez pas à vous occuper dans quel répertoire il le cherche. On utilise ces fichiers inclus principalement pour définir le contenu des bibliothèques, avec l'extension .h (header). Dans notre exemple (ligne 2) c'est iostream.h, fichier définissant les flux standard d'entrées/sorties (en anglais Input/Output stream), qui feront le lien entre le programme et la console (clavier : cin / écran : cout).

3.3) structure d'un programme simple

Dans notre petit exemple, nous en arrivons enfin à notre programme. Rappelez-vous que l'on cherche à situer chaque ligne de l'exemple dans le schéma. Nous trouvons :

- une <u>entête de fonction</u> (en ligne 4). Dans ce cas on ne possède qu'une seule fonction, la fonction principale (main function). Cette ligne est obligatoire en C/C++, elle définit le "point d'entrée" du programme, c'est à dire l'endroit où débutera l'exécution du programme.
- un "<u>bloc d'instructions</u>", délimité par des accolades {} (lignes 5 et B), et comportant :
  - des déclarations de variables (ligne 6), sous la forme : `type listevariables;`

    Une variable est un case mémoire de l'ordinateur, que l'on se réserve pour notre programme. On définit le nom que l'on choisit pour chaque variable, ainsi que son type, ici float, c'est à dire réel (type dit à virgule flottante, d'où ce nom). Nous avons donc dans cet exemple deux variables de type « float », nommée HT et TTC. Les trois types scalaires de base du C sont l'entier (int), le réel (float) et le caractère (char). On ne peut jamais utiliser de variable sans l'avoir déclarée auparavant. Une faute de frappe devrait donc être facilement détectée, à condition d'avoir choisi des noms de variables suffisamment différents (et de plus d'une lettre).

  - des instructions, dans notre exemple toutes terminées par un ;. Une instruction est un ordre élémentaire que l'on donne à la machine, qui manipulera les données (variables) du programme.

Détaillons les instructions de notre programme :

`cout<<"un texte";` affiche à l'écran le texte qu'on lui donne (entre guillemets, comme toute constante texte en C++).

`cin>>HT;` attend que l'on entre une valeur au clavier, puis la met dans la mémoire (on préfère dire variable) HT.

une **affectation** est définie par le signe =. Une affectation se fait toujours dans le même sens : on détermine (évalue) tout d'abord la valeur à droite du signe =, en faisant tous les calculs nécessaires, puis elle est transférée dans la mémoire dont le nom est indiqué à gauche du =. On peut donc placer une expression complexe à droite du =, mais à sa gauche seul un nom de variable est possible, aucune opération. En ligne 9 on commence par diviser TVA par 100 (à cause des parenthèses) puis on y ajoute 1 puis on le multiplie par le contenu de la variable HT. Le résultat de ce calcul est stocké (affecté) dans la variable cible TTC.

la ligne A affichera enfin le résultat stocké dans TTC (précédé du texte entre guillemets, et suivi d'un retour à la ligne noté « \n ».

4) Les variables simples et opérateurs associés

4.1) les types entiers

Nous allons revoir la structure d'un petit programme vue au cours précédent (et évidemment des choses nouvelles), mais présentées dans un sens différent : nous allons partir des données que nous voulons traiter pour construire le programme autour.

Un programme est un peu comme une recette de cuisine. On définit au début tous les ingrédients, plats et récipients nécessaires, puis on explique dans l'ordre exactement comment il faut procéder. J'aime aussi faire le parallèle avec une procédure, telle qu'on le comprend dans l'industrie. Ici aussi, dans un format standardisé, on définit les outils et matières brutes nécessaires, puis on détaille la manière de faire.

L'ordinateur ne sait travailler que sur des nombres en binaire. C'est pourquoi il va falloir tout codifier sous forme numérique : des nombres, mais aussi du texte, des images, du son... Limitons nous pour l'instant aux nombres. Ils seront stockés dans des mémoires de l'ordinateur, chacune définie par un numéro (appelé « adresse »). Mais à quel endroit de l'ordinateur sont placés ces nombres ne nous intéresse pas, c'est pourquoi on préfère laisser au compilateur la tâche de l'adressage des mémoires. Nous allons simplement les désigner par un nom : l'identificateur.

Quand nous aurons besoin d'une mémoire pour stocker une valeur (on nomme cela une variable), il suffit de la déclarer : on dit quel type de variable on désire, et le nom qu'on désire lui donner.

En C/C++ il existe six sortes d'entiers. Les **char** (sur 8 bits) peuvent représenter des nombres entiers de -128 à +127. Les **short** (sur16 bits) vont aller de -32768 à 32767 (32k), et les **long** de -2147483648 à 2147483647 (2G). Si jamais les valeurs négatives ne sont pas utiles, on peut utiliser les **unsigned char** (sur 1 octet) qui peuvent représenter des nombres entiers de 0 à 255. Les **unsigned short** (sur 2 octets) vont aller de 0 à 65535 (64k), et les **unsigned long** de 0 à 4G. Ces types (précis) sont utiles si l'on cherche à gagner de la mémoire (dans le cas de nombreuses variables) ou que l'on tente de faire correspondre un nombre exact de bits avec un certain nombre d'entrées ou sorties (en informatique industrielle). Certains compilateurs acceptent également des **long long** sur 64 bits.

Mais la plupart du temps, on utilisera le type **int,** qui sera choisi par le compilateur de manière à être le plus efficace possible, tout en permettant de gérer au minimum les entiers entre -32768 et 32767 (Turbo C++ il prend des short, gcc prend des long).

Attention, le compilateur ne prévoit pas de tester les dépassements de capacité, c'est à vous de le faire. Si jamais vous dépassez le plus grand entier positif prévu par le type que vous avez choisi, il repart du plus petit ! Donc pour un char, après 127 il considère qu'il y a -128, pour un unsigned short après 65535 il repasse à 0.

4.2) Expressions, affectation

Une expression est un calcul qui donne une valeur résultat (exemple : 8+5). Une expression comporte des variables, des appels de fonction et des constantes combinés entre eux par des opérateurs (ex : `MaVariable*sin(VarAngle*PI/180)`).

Une expression de base peut donc être un appel à une fonction (exemple sin(3.1416). Une fonction est un bout de programme (que vous avez écrit ou faisant partie d'une bibliothèque) auquel on "donne" des valeurs (arguments), entre parenthèses et séparés par des virgules. La fonction fait un calcul sur ces arguments pour

"retourner" un résultat. Ce résultat pourra servir, si nécessaire, dans une autre expression, voire comme argument d'une fonction.

Dans une expression, on peut imposer l'ordre des calculs à l'aide des parenthèses (sans parenthèses, c'est le compilateur qui choisit l'ordre, d'après des règles bien précises et connues des programmeurs).

Une expression peut aussi comporter une **affectation**. Elle est utilise le signe =, et signifie que l'on veut mettre un nombre dans une variable. L'ordinateur détermine (évalue) tout d'abord la valeur à droite du signe =, en faisant tous les calculs nécessaires, puis elle est transférée dans la mémoire dont le nom est indiqué à gauche du =. Si nécessaire, le résultat d'une affectation est la valeur qu'on a transférée dans la mémoire. Cela n'a pas grand chose à voir avec l'égalité mathématique :

- on peut écrire `a=5` mais pas `5=a`
- on ne peut pas écrire `a+b=0`
- `a=a+1` signifie : regarder combien vaut a, ajouter 1, puis stocker le résultat dans a (son ancienne valeur étant alors écrasée par la nouvelle).
- `a=(b=5)+1` calcule d'abord l'intérieur de la parenthèse (met 5 dans b, cela vaut la valeur transférée donc 5), auquel on ajoute 1, on transfère donc 6 dans a (b reste à 5).

4.3) opérateurs sur les entiers

Il existe 5 opérations sur les entiers. Ce sont + (addition), - (soustraction), * (produit), / (division), % (reste de la division). Si vous ne mettez pas de parenthèses, * et / ont une priorité supérieure à + et -. Attention, pour les multiplications l'étoile est obligatoire (a*b et pas ab). Le résultat de ces opérations est toujours du même type que ces deux arguments, quand ils ont du même type (s'ils sont d'un type différent, voir plus loin). Donc 11/3 donnera 3, et pas 3,6666. 11%3 donnera 2. C'est comme à l'école primaire : 11 divisé par 3 donne 3 reste 2.

4.4) les flottants

En informatique, on n'arrive pas à gérer tous les nombres réels. On ne peut que les approximer par une suite finie de chiffres. En fait l'ordinateur gère les réels avec une méthode proche de la notation scientifique : 123,45 = 1,2345 10^2 (noté 1.2345E2 en C). Ce qui différenciera donc les types de nombres réels sera le nombre de chiffres significatifs pris en compte. Attention, l'ordinateur utilise des chiffres en binaire, c'est pourquoi l'indication d'un nombre de chiffres en décimal sera approximative. Il existe trois types de réels, les **float** (jusqu'à 1,7E38 mais avec uniquement 7 chiffres décimaux significatifs), les **long float** ou **double** (jusqu'à plus de 1E300, avec 15 chiffres significatifs, et les **long double** (non normalisés) encore plus précis.

Les 4 opérations possibles sont +, -, *, / (ici la division donnera un flottant, il n'y a pas de reste).

4.5) le cast

Lorsque, dans une expression, les deux opérandes sont de type différent (mais numérique évidement), le compilateur prévoit une conversion implicite (vous ne l'avez pas demandée mais il la fait néanmoins) suivant l'ordre : { char -> short -> long -> float -> double } et { signed -> unsigned }. le « plus petit » est automatiquement transformé dans le type du « plus grand », le résultat de l'opération sera lui aussi du type du plus grand. Par contre, le cast n'est appliqué automatiquement que si c'est nécessaire : si un opérateur est entouré de short, l'opération est faite sur des short, le résultat (short) pouvant toujours être transformé plus tard si nécessaire.

Pour forcer un cast, il suffit d'indiquer le type désiré entre parenthèses, devant la valeur à transformer (comme vous placez le signe - devant un nombre pour lequel vous désirez changer le signe). Exemple : (float)3 vaut 3 en flottant (vous pouviez aussi directement écrire 3.0). Vous pouvez également forcer un cast vers un type « plus petit » : dans ce cas là il y a risque de perte d'information. Exemple : (int)3.9 donnera 3 (partie entière), (unsigned char)256 donnera 0 (j'ai déjà dit qu'après le dernier il repassait à 0).

exemples :

`int a=64,b=2;`	
`float x=1,y=2;`	
`b=7/b;`	/* signe = donc en premier calcul de l'argument à droite : 7 (entier) / 2 (entier) donne 3 (entier, reste 1, que l'on obtiendrait par 5%2). donc b=3 */
`x=7/b;`	/* 7 et b entiers => passage en réel inutile, calcul de 7/3 donne 2 (entier, reste 1) puis opérateur = (transformation du 2 en 2.0 puis transfert dans X qui vaut donc 2.0) */
`x=7/y;`	/* un int et un float autour de / : transformation implicite de 7 en réel (7.0), division des deux réel (3.5), puis transfert dans x */
`x=7.0/b;`	/* un int et un float autour de / : transformation implicite de b en réel, division des deux réel (3.5), puis transfert dans x */
`x=((float)(a+1))/b;`	/* calcul (entier) de a+1, puis transformation explicite en float, et donc implicite de b en float, division 65.0/3.0 -> 21.666... */

4.5) retour sur la structure d'un petit programme

Donc refaisons un nouveau programme. Lorsque, dans notre commerce, nous vendons plusieurs objets d'un même prix unitaire, nous aimerions connaître le prix total. De quelles variables aurons nous besoin ? Le prix unitaire et le prix total sont des flottants (si l'on a des prix avec des centimes). Nommons les PU et PTot. Par contre le nombre d'objets est entier, nommons le N.

Comme nous allons utiliser le clavier et l'écran, il nous faut inclure iostream.h. Puis nous allons passer au programme, il faut y mettre son entête et l'encadrer entre accolades. Nous commençons par spécifier (on dit déclarer) les variables nécessaires. Puis il faut demander à l'utilisateur de notre programme le prix unitaire et le nombre d'objets. On pourra alors calculer le prix total (n'oubliez pas l'ordre : en premier, l'ordinateur calcule combien vaut l'expression à droite du signe =, puis met ce nombre dans la variable identifiée à gauche du =). Il ne faudra pas oublier de l'afficher à l'écran, car à la fin du programme toutes les variables seront libérées, comme à la fin d'une recette de cuisine on lave et range le matériel.

```
#include <iostream.h>
int main(void)
 {
  float PU, PTot;
  int N;
  cout<<"veuillez entrer le prix unitaire : ";
  cin>>PU;
  cout<<"veuillez entrer le nombre d'objets : ";
  cin>>N;
  PTot=PU*N;
  cout<<"prix total : "<<PTot<<"\n";
 }
```

5) Les instructions et leur séquencement

5.1) définitions

Une instruction peut être :

- soit une expression (en fait, le plus souvent un calcul avec une affectation, un appel de fonction, l'application d'une méthode sur un objet...), terminée par un « ; » qui en fait signifie « on peut oublier le résultat de l'expression et passer à la suite »,
- soit une structure de contrôle : boucle, branchement... (détaillé ci-après),
- soit un bloc d'instructions : ensemble de déclarations et instructions délimités par des accolades {}. Un bloc sera utilisé à chaque fois que l'on désire mettre plusieurs instructions là où on ne peut en mettre qu'une.

Les instructions seront toujours exécutées dans l'ordre où elles ont été écrites, l'une après l'autre. Ce n'est qu'à l'intérieur des structures de contrôle que l'on peut modifier leur séquencement (leur ordre d'exécution), en permettant de refaire plusieurs fois certaines instructions (boucles), de passer directement d'un endroit à un autre du programme (branchements inconditionnels), de n'exécuter certaines instructions que dans certains cas (branchements conditionnels).

5.2) précision sur le « ; »

Le « ; » sert à indiquer la fin d'une instruction (rappel, le retour à la ligne n'indique rien de plus qu'un simple espace).

La première forme d'instruction est nécessairement terminée par un « ; ». Voici quelques exemples :

```
a=5;                                //affectation simple
x=4+2*(sin(3*PI)-tan(z));      //avec appel de fonctions
cout<<"le résultat est"<<x<<" pour "<<y<<"\n"      //E/S
lancer_sous_tache(a,b,c);         //appel sous-programme
MonObjet.dessiner();         //application d'une méthode
```

Un cas particulier est l'instruction vide, qui se compose uniquement d'un ; (utilisé là où une instruction est nécessaire d'après la syntaxe).

Dans la seconde forme (structure de contrôle), je préciserai à chaque fois la position du « ; ».

Un bloc quand à lui n'a pas besoin d'être suivi d'un «; », puisque l'accolade fermante délimite déjà la fin du bloc (par contre les instructions comprises à l'intérieur d'un bloc devront peut-être se terminer par un «; », même celle juste avant l'accolade fermante).

6) Structures de contrôle : les boucles

Une boucle permet de répéter plusieurs fois un bloc d'instructions.

6.1) boucle while (tant que)

structure : **`while (expression) instruction`**

Tant que l'expression est vraie, on effectue l'instruction, qui peut être simple (terminée par ;), bloc (entre {}) ou vide (; seul). Il ne faut donc surtout pas de ; derrière la), puisque dans ce cas seule l'instruction vide est répétée plusieurs fois, l 'instruction suivante ne sera faite qu'une seule fois. L'expression est au moins évaluée une fois (évaluer signifie « calculer la valeur »). Tant que l'expression est vraie, on effectue l'instruction, dès qu'elle est fausse, on passe à l'instruction suivante (si elle est fausse dès le début, l'instruction n'est jamais effectuée).

exemple :

```
#include <iostream.h>
int main(void)
 {
  float nombre,racine=0;
  cout<<"entrez un nombre réel entre 0 et 10";
  cin>>nombre;
  while (racine*racine<nombre) racine+=0.01;
  cout<<"la racine de"<<nombre<<"vaut"<<racine<<"à 1\% près\n";
 }
```

L'expression est le plus souvent formée de tests utilisant les opérateurs < (inférieur), > (supérieur), <= (inférieur ou égal), >= (supérieur ou égal), = = (égal, attention un seul signe = signifie affectation !), != (différent), qui peuvent être combinés par les opérateurs booléens && (et), || (ou) et ! (non). Exemples d'expressions valides :

```
        (a<100)
        (a>0 && a<=10)
        ((a>0 && b>0) || c==0)
```

A la rigueur, l'expression peut être numérique, auquel cas on considère qu'elle est fausse si elle vaut 0, et qu'elle est vraie dans tous les autres cas.

Exercice (while_puiss) : faire un programme qui affiche toutes les puissances de 2, jusqu'à une valeur maximale donnée par l'utilisateur. On calculera la puissance par multiplications successives par 2.

Exercice (while_err) : que fait ce programme ?

```
#include <iostream.h>
#include <math.h>
#define debut 100
#define pas 0.01
int main(void)
 {
  float nombre=debut;
  int compte=0,tous_les;
  cout<<("afficher les résultats intermédiaires tous les ? (333 par exemple) ?");
  cin<<tous_les;
  while (fabs(nombre-(debut+(compte*pas)))<pas)
   {
    nombre+=pas;
    if (!(++compte%tous_les))
       cout<<"valeur obtenue"<<nombre<<", au lieu de"
       <<(float)(debut+(compte*pas))
       <<"en"<<compte<<"calculs\n";
   }
  cout<<"erreur de 100\% en"<<compte<<"calculs\n";
 }
```

6.2) boucle do while (faire tant que)

structure : **`do instruction while (expression);`** (attention au ; final)

Comme pour while, l'instruction est répétée tant que l'expression est vraie. Mais quoi qu'il arrive, l'instruction est au moins faite une fois, avant la première évaluation de l'expression.

exemple :

```
#include <iostream.h>
int main(void)
 {
  int a;
  do
   {
    cout<<"entrez le nombre 482";
    cin>>a;
   }
  while (a!=482);
  cout<<"c'est gentil de m'avoir obéi\n";
 }
```

Exercice (do_while) : écrivez un programme de jeu demandant de deviner un nombre entre 0 et 10 choisi par l'ordinateur. On ne donnera pas d'indications avant la découverte de la solution, où l'on indiquera le nombre d'essais. La solution sera choisie par l'ordinateur par la fonction rand() qui rend un entier aléatoire.

6.3) boucle for (pour)

structure : **`for (expr_initiale; expr_condition; expr_incrémentation) instruction`**

Cette boucle est surtout utilisée lorsque l'on connaît à l'avance le nombre d'itération à effectuer. L'expr_initiale est effectuée une fois, en premier. Puis on teste la condition. On effectue l'instruction puis l'incrémentation tant que la condition est vraie. L'instruction et l'incrémentation peuvent ne jamais être effectuées. La boucle est équivalente à :

```
    expr_initiale;
    while (expr_condition)
     {
      instruction
      expr_incrémentation;
     }
```

On peut remarquer qu'en C/C++ on compte en commençant par 0. Pour dire 10 fois bonjour, on écrira :

```
for(i=0;i<10;i++)cout<<"bonjour\n";
```

En fait, pendant qu'il dit la première fois bonjour, i vaut 0 (car il n'a pas encore fini la première fois). Puis, quand il a effectivement dit bonjour, i passe à 1 et il dit pour la seconde fois bonjour. Et ceci tant que i est strictement inférieur à 10 (donc jusqu'à 9 compris). Dès que i passe à 10, la boucle est arrêtée, et on passe à la suite.

Une ou plusieurs des trois expressions peuvent être omises, l'instruction peut être vide. for(;;); est donc une boucle infinie.

autre exemple :

```
{ char c; for(c='Z';c>='A';c--)cout<<c; }
```

Exercice (for) : faire un programme qui calcule la moyenne de N notes. N et les notes seront saisies par cin. Le calcul de la moyenne s'effectue en initialisant une variable à 0, y ajoutant progressivement les notes saisies puis division par N).

7) Structures de contrôle : les branchements conditionnels

On a souvent besoin de n'effectuer certaines instructions que dans certains cas. On dispose pour cela du IF et du SWITCH.

7.1) If - Else (Si - Sinon)

structure : **`if (expression) instruction1`**
ou : **`if (expression) instruction1 else instruction2`**

Si l'expression est vraie on effectue l'instruction1, puis on passe à la suite. Sinon, on effectue l'instruction 2 puis on passe à la suite (dans le cas sans else on passe directement à la suite).

Exercice (jeu) : modifier le jeu de l'exercice (do_while) en précisant au joueur à chaque essai si sa proposition est trop grande ou trop petite.

L'instruction d'un if peut être un autre if (imbriqué). Exemple :

```
if(c1) i1;
else if (c2) i2;
else if (c3) i3;
else i4;
i5;
```

Explication : si c1 est vrai alors on exécute i1 puis i5, sinon mais si c2 alors i2 puis i5, ... Si ni c1 ni c2 ni c3 alors i4 puis i5.

Le else étant facultatif, il peut y avoir une ambiguïté s'il y a moins de else que de if. En fait, un else se rapporte toujours au if non terminé (c'est à dire à qui on n'a pas encore attribué de else) le plus proche. Une solution pour lever l'mbiguïté est de délimiter le if sans else en l'entourant de {}.

exemples :

- `if(c1) if(c2) i1; else i2;`
  si c1 et c2 alors i1, si c1 et pas c2 alors i2, si pas c1 alors (quel que soit c2) rien.
- `if (c1) {if (c2) i1;} else i2;`
  si c1 et c2 alors i1, si c1 et pas c2 alors rien, si pas c1 (quel que soit c2) alors i2.

7.2) Switch - Case (brancher - dans le cas)

Le switch est aussi appellé le « goto calculé » Cette structure de contrôle permet de remplacer une suite trop longue de if et else if (mais rien ne vous force à l'utiliser, vous pouvez parfaitement vous limiter aux ifs).

```
structure :
        switch(expression_entière)
```

```
{
        case cste1:instructions
        case cste2:instructions
                ........
        case csteN:instructions
        default :instructions
}
```

L'expression ne peut être qu'entière (char, int, long). L'expression est évaluée, puis on passe directement au "case" correspondant à la valeur trouvée. Le cas default est facultatif, mais si il est prévu il doit être le dernier cas.

exemple : fonction vérifiant si son argument c est une voyelle.

```
int voyelle(char c)
 {
  switch(c)
   {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'y':return(1); /* 1=vrai */
    default :return(0)
   }
 }
```

Remarque : l'instruction break permet de passer directement à la fin d'un switch (au }). Dans le cas de switch imbriqués on ne peut sortir que du switch intérieur.

exemple :

```
switch (a)
   {
    case 1:inst1;inst2;....;break;
    case 2:....;break;
    default:.....
   } /*endroit où l'on arrive après un break */
```

Exercice (calcul) : faire un programme simulant une calculatrice 4 opérations.

8) Branchements inconditionnels (goto)

Nous en finissons ici avec les structures de contrôle. Ces instructions sont certes utiles, mais moins importantes, c'est pourquoi je ne les ai pas présentées en cours (peut-être s'il reste du temps en fin de semestre ?). Quand on arrive sur une telle instruction, on se branche obligatoirement sur une autre partie du programme. Ces instructions sont à éviter si possible, car elles rendent le programme plus complexe à maintenir, le fait d'être dans une ligne de programme ne suffisant plus pour connaître immédiatement quelle instruction on a fait auparavant, et donc ne permet plus d'assurer que ce qui est au dessus est correctement terminé. Il ne faut les utiliser que dans certains cas simples.

8.1) goto (aller à)

La pratique des informaticiens a montré que l'utilisation des goto donne souvent des programmes non maintenables (impossibles à corriger ou modifier). Les problèmes qu'ils posent ont amené les programmeurs expérimentés à ne s'en servir qu'exceptionnellement.

structure : **goto label;**

Label est un identificateur (non déclaré, mais non utilisé pour autre chose), suivi de deux points (:), et indiquant la destination du saut. Un goto permet de sortir d'un bloc depuis n'importe quel endroit. Mais on ne peut entrer dans un bloc que par son { (qui créera proprement les variables locales du bloc).

```
{.....
 {.....
  goto truc;
  .....
 }
.....
truc:
.....
}
```

Les goto sont néanmoins acceptables si la destination du saut est un lieu « normal » de sortie (fin de boucle, de fonction...). C'est pourquoi on a créé les quatre instructions spécifiques ci après.

8.2) break (interrompre)

Il provoque la sortie immédiate de la boucle ou switch en cours. Il est limité à un seul niveau d'imbrication.

exemples :

```
do {if(i==0)break;....}while (i!=0);  /* un while aurait été mieux */
for (i=0;i<10;i++){....;if (erreur) break;} /* à remplacer par for(i=0;(i<10)&&
(!erreur);i++){...} */
```

8.3) continue (continuer)

Cette instruction provoque le passage à la prochaine itération d'une boucle. Dans le cas d'un while ou do while, on saute vers l'évaluation du test de sortie de boucle. Dans le cas d'un for on passe à l'expression d'incrémentation puis seulement au test de bouclage. En cas de boucles imbriquées, il permet uniquement de continuer la boucle la plus interne.

exemple :

```
for (i=0;i<10;i++) {if (i==j) continue; ...... }
```

peut être remplacé par

```
for (i=0;i<10;i++) if (i!=j) { ...... }
```

8.4) return (retourner)

Permet de sortir de la fonction actuelle (y compris main), en se branchant à son dernier }. Return permet également (et surtout) de rendre la valeur résultat de la fonction.

structure: **return;** ou **return(valeur);**

exemple :
```
int max(int a, int b) {if (a>b) return(a); else return(b);}
```

8.5) exit (sortir)

Ceci n'est pas un mot clef du C mais une fonction disponible dans la plupart des compilateurs (définie par ANSI, dans stdlib.h). Elle permet de quitter directement le programme (même depuis une fonction). On peut lui donner comme argument le code de sortie (celui que l'on aurait donné à return dans main). Cette fonction libère la mémoire utilisée par le programme (variables + alloc) et ferme (sur beaucoup de compilateurs) les fichiers ouverts.

```
structure : exit() ou exit(code)
```

9) Fonctions

Quand un programme devient important, il devient également compliqué à gérer, du fait de son grand nombre de lignes et surtout des imbrications multiples des structures de contrôle. C'est pourquoi il est nécessaire de structurer le programme. Le plus efficace est de le découper en différentes parties, chacune effectuant une tâche particulière. Ce sont les différents sous-programmes ou procédures, que l'on appelle « fonctions » en C. Il va donc falloir, avant d'écrire un programme, essayer de le décomposer en sous-tâches, avec pour chacune une définition des données à lui fournir, les traitements à y appliquer, les données résultantes attendues. Prenons pour exemple un technicien sur machine outil. On peut détailler précisément les différentes tâches qu'il peut effectuer (ce sont les fonctions). Et à un instant donné, on lui donne la liste des tâches à effectuer dans la journée (c'est la fonction main).

9.1) Définition générale

Une fonction est définie par sont entête, suivie d'un bloc d'instructions (voir schéma au cours 1). L'entête est de la forme :

`type_retourné nom_fonction(liste_arguments_formels)` (surtout pas de ;)

9.2) procédure (fonction ne retournant rien)

Si la fonction ne retourne rien, le type retourné est void. De même si la fonction n'utilise pas d'arguments il faut indiquer void à la place de la liste d'arguments.

regardons cet exemple :

```
void dire_trois_fois_bonjour(void)
{
 int i;
 for(i=0;i<3;i++) cout<<"bonjour\n";
}
```

Cette fonction effectue une tâche (dire trois fois bonjour). Pour cela, elle n'a besoin d'aucune information spécifique (on dit qu'elle ne reçoit pas d'arguments), et elle n'a rien non plus à retourner. C'est la raison des deux « void » dans l'entête. Toutes les variables déclarées dans le bloc sont locales (ici, c'est "i"). Ceci signifie qu'elles existent dans la fonction, et uniquement pendant qu'on traite cette fonction. A la sortie (dernière }), la mémoire sera libérée. En clair, la variable i de notre fonction n'est pas la même que celles qui pourraient s'appeler aussi i mais qui seraient déclarées dans d'autres blocs.

L'écriture du texte ci-dessus ne fait que définir la fonction. C'est à dire qu'on a appris au compilateur comment on disait trois fois bonjour, mais on ne lui a pas encore dit quand le faire. Ceci se fait dans le déroulement séquentiel des instructions d'une fonction, par l'appel de la fonction qui se note :

```
nom_fonction(liste_arguments_réels);
```

ici : dire_trois_fois_bonjour(); que l'on pourrait par exemple placer dans la fonction « main ».

Pour appeler une fonction, il faut qu'elle ait été définie auparavant, soit parce qu'elle a été écrite avant la procédure qui l'appelle, soit parce qu'on l'a « prototypée », ce qui se fait en indiquant son entête, suivie d'un ; (j'en reparlerai un peu plus loin).

Prenons l'exemple de notre technicien. La tâche "nettoyer la machine" ne nécessite aucune matière première spécifique, c'est pourquoi il n'y a pas de transmission d'arguments. Par contre il a besoin pour cette tâche d'utiliser divers outils, qu'il va trouver sur son poste spécifiquement pour cela (une soufflette par exemple), ce sont les variables locales.

Voyons désormais une fonction recevant des arguments. Par exemple :

```
void dire_n_fois_bonjour(int N)
{
 int i;
 for(i=0;i<N;i++) cout<<"bonjour\n";
}
```

Cette fonction a besoin d'une information : le nombre de bonjours désirés (c'est un entier). On l'appelle N dans la fonction, on l'appelle argument formel. Au niveau de l'appel, il faudra donner un entier (l'argument réel) qui peut être une variable (qui n'a pas besoin d'avoir le même nom), mais aussi une constante ou toute expression donnant un entier :

```
dire_n_fois_bonjour(10);
int j=5;
dire_n_fois_bonjour(j);
dire_n_fois_bonjour(j*2-8);
```

On peut également transmettre plus d'un argument à une fonction. Mais il faudra qu'à l'appel, on fournisse le même nombre d'arguments réels, dans le même ordre.

Pour notre technicien, la tâche "vidanger la machine" nécessite qu'on fournisse de la matière première : un bidon d'huile. C'est l'argument fourni à la tâche.

9.3) fonction retournant une valeur

Passons maintenant aux fonctions retournant une valeur. Voyons cet exemple :

```
float produit(float a;float b)
 {
  float z;
```

```
  z=a*b;
  return(z);
 }
```

Cette fonction nécessite qu'on lui fournisse deux flottants. Elle les appelle a et b (mais ils n'ont pas besoin d'avoir le même nom à l'appel). Elle fait un calcul (ici certes trop simple pour justifier l'utilisation d'une fonction), puis quand elle a terminé, elle retourne la valeur de z. Au niveau de l'appel, il faut bien évidemment fournir deux valeurs (arguments réels) à la fonction. Mais surtout il faut utiliser la valeur retournée par la fonction, par une affectation, une opération d'entrée-sortie, ou même en argument d'une autre fonction :

```
x=produit(z,3*y);
cout<<produit(x,produit(3,2));
for(i=0;sin(z)<produit(x,i);i++);
```

Pour notre technicien, la tâche "assemblage de la pièce" nécessite des arguments : les différents composants nécessaires. Peut-être aura-t-il besoin également de variables locales (outils spécifiques). Mais à la fin de la tâche nous exigeons qu'il nous donne (en C on dit retourne) la pièce assemblée. Attention, une fonction peut soit ne rien retourner (void), soit retourner une seule valeur. Pour en retourner plusieurs il faut utiliser divers artifices, comme regrouper toutes les valeurs dans un seul "objet". Ou alors on utilise le passage d'arguments par adresse.

9.4) passage d'arguments (par valeur, par référence,...)

Pour l'instant , nous avons utilisé un passage d'arguments par valeur (comportement par défaut en C).

Imaginons la fonction :

```
void échange(int i;int j)
 {
  int tampon;
  tampon=i;
  i=j;
  j=tampon;
 }
```

Lors d'un appel à cette fonction par échange(x,y), les variables locales i,j,tampon sont créées localement. i vaut la valeur de x, j celle de y. Les contenus de i et j sont échangés puis la pile (endroit qui contient toutes les variables locales) est libérée, sans modifier x et y. Pour résoudre ce problème, il faut utiliser un passage d'arguments par référence. On définira la fonction ainsi :

```
void échange(int &i;int &j) //le reste de la fonction (le bloc) est inchangé
```

On appelle la fonction par `échange(x,y);` Les deux arguments formels de la fonction (i et j) sont des références, c'est à dire qu'au lieu d'avoir transmis à la fonction la valeur d'x et y (combien ils valent), on lui a transmis leur adresse (où ils sont). La fonction peut maintenant les modifier, puisqu'elle sait où sont stockés en mémoire les arguments formels.

On utilise le passage d'arguments par référence quand on désire qu'un argument réel puisse être modifié par la fonction. Mais on peut aussi les utiliser lorsque l'on désirerait retourner plusieurs valeurs : il suffit de donner à la fonction les adresses de différentes variables dans lesquelles la fonction écrira les résultats qu'elle aimerait nous retourner.

On s'en sert également si l'on désire éviter une copie d'un objet de grande taille de l'argument réel vers l'argument formel, même si on ne désire pas modifier l'argument. Pour cela, on peut ajouter le mot clef "const" devant le type, dans l'entête, pour plus de sécurité.

Comme en C ANSI le passage par référence n'était pas possible, on utilisait un passage par adresse (pointeurs). Cela reste possible, mais peut-être plus complexe. Je n'en dirai donc pas plus ici.

9.5) appel de fonctions, prototypes

Les arguments (formels) sont des variables locales à la fonction. Les valeurs fournies à l'appel de la fonction (arguments réels) y sont recopiés à l'entrée dans la fonction. Les instructions de la fonction s'exécutent du début du bloc ({) jusqu'à return(valeur) ou la sortie du bloc (}). La valeur retournée par la fonction est indiquée en argument de return. Toutes les variables locales (arguments formels et variables déclarées dans le bloc) sont libérées au retour de la fonction.

Si l'on désire appeler une fonction, il faut l'avoir définie avant. Soit on les définit dans le bon ordre (donc en particulier on définit "main" en dernier), soit on utilise un prototype : c'est une déclaration (en général globale, souvent placée avant la première définition de fonction). On y indique les mêmes informations que dans l'entête : le type retourné, le nom de la fonction, et le type des arguments. Seul le nom des arguments peut être omis. Un prototype est toujours suivi d'un « ; ».

Le prototype permet aussi de définir des valeurs d'arguments par défaut. Les arguments réels peuvent être omis en commençant par le dernier (impossible d'omettre les premiers si l'on veut préciser un suivant).

```
void maFonction(int argument1=18,int argument2=3);
maFonction(1,2); //c'est moi qui fixe tout
maFonction(10);  //le deuxième argument vaudra 3
maFonction();    //les deux sont fixés par défaut
```

9.6) Exemple

Nous pouvons maintenant écrire tout un programme, décomposé en fonctions. Je vous rappelle que la fonction main constitue le point d'entrée du programme, c'est à dire la fonction appelée en premier. Dans cet exemple, j'ai également placé une variable globale (bien que ce soit une très mauvaise pratique).

```
#include <iostream.h>
void affiche_calcul(float,float); /* prototype */
float produit(float,float);
void message(void);
int varglob; //ceci est une variable globale accessible partout et tout le temps
int main(void)
 {
  float a,b; /* déclaration locale */
  message();
  varglob=0;
  cout<<("veuillez entrer 2 valeurs : ");
  cin>>a;
  cin>>b;
  affiche_calcul(a,b);
  cout<<"nombre d'appels à produit : "<<<"\n";
 }
float produit(float r, float s)
 {
  varglob++;
  return(r*s);
 }
void affiche_calcul(float x,float y)
```

```
 {
  float varloc;
  varloc=produit(x,y);
  varloc=produit(varloc,varloc);
  cout<<"le carré du produit est "<<<"\n";
 }
void message(void)
 {
  const char ligne[]="******************************\n";
  cout <<<"bonjour\n"
 }
```

9.7) Récursivité, gestion de la pile

Nous n'avons pas traité la récursivité en cours. Bien que ce soit une notion tres importante, il faut déjà une certaine expérience pour la comprendre, ce qui n'est pas encore le cas à ce niveau du cours (peut-être après quelques TP ?)

Une fonction peut s'appeler elle-même :

```
int factorielle(int i)
 {
  if (i>1) return(i*factorielle(i-1));
  else return(1);
 }
```

analysons l'état la pile lors d'un appel à factorielle(3) :

		i=1		
	i=2	i=2	i=2	
i=3	i=3	i=3	i=3	i=3
(a)	(b)	(c)	(d)	(e)

- (a) appel de factorielle(3), création de i, à qui on affecte la valeur 3. comme i>1 on calcule i*factorielle(i-1) : i=3,i-1=2 on appelle factorielle(2)
- (b) création i, affecté de la valeur 2, i>1 donc on appelle factorielle(1)
- (c) création de i, i=1 donc on quitte la fonction, on libère le pile de son sommet, on retourne où la fonction factorielle(1) a été appelée en rendant 1.
- (d) on peut maintenant calculer i*factorielle(1), i (sommet de la pile) vaut 2, factorielle(1) vaut 1, on peut rendre 2, puis on "dépile" i
- (e) on peut calculer i*factorielle(2), i vaut 3 (sommet de la pile), factorielle(2) vaut 2, 3*2=6, on retourne 6, la pile est vidée et retrouve sont état initial.

Attention, la récursivité est gourmande en temps et mémoire, il ne faut l'utiliser que si l'on ne sait pas facilement faire autrement :

```
int factorielle(int i)
 {
  int result;
  for(result=1;i>1;i--) result*=i;
  return(result);
 }
```

10) Objets

10.1) introduction

Nous allons désormais aborder une notion fondamentale : l'objet. L'objet n'existe pas en C standard, il est le principal apport de C++. L'objet est un groupement de données (même de types différents) appelées « attributs » et des fonctions associées (appelées « méthodes »). Nous pourrions prendre comme exemple une date, qui contient trois attributs jour, mois et année. Les méthodes associées pourraient être (entre autres) une fonction permettant de déterminer le jour de la semaine (le 8/11/04 est un lundi), une fonction permettant de comparer des dates, etc...

Nous allons pour ce qui suit utiliser l'exemple des vecteurs.

10.2) la classe

Comme toujours en C++, il faut avant tout déclarer les objets avant de les utiliser. C'est ce que l'on fait en déclarant une « classe ». La définition d'une classe permet d'ajouter un nouveau type à ceux déjà connus par le compilateur (comme les entiers ou les flottants). Définissons donc la classe « vecteur ». Un vecteur comporte trois composantes (à valeur réelle) : ce sont les attributs d'un vecteur. Nous les appellerons x, y et z. Puis nous définirons les méthodes utilisées pour les vecteurs : comment on les additionne, on calcule leur norme, la multiplication par un réel, le produit scalaire et vectoriel... Mais aussi comment on les affiche et les saisit.

```
class vecteur
 {
   //définition des attributs
   float x,y,z;
   //déclaration des méthodes
   void saisir(void);
   void afficher(void)
   float norme(void);
   void produit(float);
   void additionner(vecteur);
   //arrêtons nous là pour l'instant
 };
```

Remarquez le point virgule qui termine la déclaration (comme toute autre déclaration d'ailleurs). C'est un des rares cas où une accolade fermante est suivie d'un point virgule.

10.3) définition des méthodes

Nous n'avons que déclaré les prototypes des méthodes que nous prévoyons d'utiliser avec nos vecteurs. Nous aurions pu les définir complètement dans la classe (entre accolades), mais nous allons plutôt utiliser une définition à l'extérieur de la classe. Pour cela, il faut utiliser une entête de fonction spéciale, spécifiant la classe à laquelle la méthode s'appliquera :

type_retourné nom_classe::nom_méthode(arguments)

Définissons pour commencer l'affichage d'un vecteur : il suffit d'afficher ses trois coordonnées (nous les mettrons entre crochets) :

```
void vecteur::afficher(void)
```

```
{
 cout<<"["<<x<<","<<y<<","<<z<<"]";
}
```

Vous pouvez remarquer que cette fonction n'a besoin d'aucun argument. En effet, toute méthode peut directement accéder aux attributs de l'objet auquel elle s'applique. La connaissance des trois attributs x, y et z suffit à notre méthode, c'est pourquoi elle n'a besoin d'aucun argument supplémentaire (void). De même, une fois son affichage terminé, elle n'a aucune valeur particulière à retourner, d'où son type void. On définirait de la même manière la saisie :

```
void vecteur::saisir(void)
 {
  cout<<"x?";cin>>x;
  cout<<"y?";cin>>y;
  cout<<"z?";cin>>z;
 }
```

Passons maintenant à la norme. La norme d'un vecteur est un réel, que l'on calcule uniquement à partir de ses coordonnées, nous n'avons besoin de rien de plus. C'est pourquoi cette méthode ne reçoit aucun argument, mais retourne un float :

```
float vecteur::norme(void)
 {float r;
  r=sqrt(x*x+y*y+z*z);
  return r;
 }
```

Pour multiplier un vecteur par un flottant, il faut (en plus des attributs donc des coordonnées du vecteur) connaître ce flottant. Si c'est le vecteur auquel s'applique la méthode qui est directement modifié par cette méthode, il n'y a rien à retourner. Ce qui nous donne (x*=a est équivalent à x=x*a) :

```
void vecteur::produit (float a)
 {
  x*=a; y*=a; z*=a;
 }
```

Nous n'allons pas encore définir l'addition pour l'instant, il va falloir attendre un peu.

10.4) utilisation d'un objet

A tout endroit où l'on peut déclarer une variable, on peut désormais déclarer vecteur. Par exemple :

```
vecteur V1, V2;
```

à partir de là, on accède à un attribut d'un objet par « objet.attribut », et à une méthode par « objet.méthode (arguments) », du moins quand on a le droit d'y accéder. En effet, par défaut, seuls les objets d'une même classe peuvent accéder aux attributs et méthodes. Donc là où on y a accès; on peut appliquer la méthode « saisir » à v1, en écrivant :

```
v1.saisir();
```

puis on peut l'afficher par :

```
v1.afficher();
```

ou calculer sa norme :

```
n=v1.norme();
```

Nous allons maintenant pouvoir définir l'addition. Nos désirons additionner un second vecteur à un premier. On appellera cette méthode par exemple par :

```
v1.additionner(v2);
```

ce qui signifie qu'on va appliquer l'addition de v2 au vecteur v1. C'est donc v1 qui sera modifié, alors que v2 correspond aux informations supplémentaires que l'on transmet à la méthode pour pouvoir effectuer sa tâche. Quand on parle des attributs x, y et z, on parle bien de ceux de l'objet à qui s'applique la méthode, ici v1.

```
void vecteur::additionner(vecteur arg)
 {
  x+=arg.x y*=arg.y; z*=arg.z;
 }
```

Comme pour toute fonction, dans la méthode l'argument a un nom « formel » : on l'appelle arg, Mais il correspondra à un argument réel qui dépend de la manière dont on a appelé la méthode (ici c'est v2)

10.5) accessibilité aux membres d'une classe

On peut définir trois degrés d'accessibilité aux membres (arguments et méthodes) d'une classe :

- private (privé) : seuls les autres membres de la même classe peuvent y accéder. C'est l'accès par défaut pour une classe.
- protected (protégé) : accès réservé aux membres de la classe, mais également aux « héritiers » (voir plus loin)
- public : même accessibilité que toute déclaration globale

il suffit de définir le type d'accès désiré, il s'appliquera à toutes les déclarations qui suivront.

En général, il vaut mieux définir de manière privée les attributs d'une classe (c'est l'accès par défaut). En effet, l'intérêt de l'approche « objet » est qu'on n'a pas à accéder à l'organisation interne des données, il nous suffit d'accéder aux méthodes. Dans notre exemple du vecteur, qui comporte trois coordonnées, on n'a pas besoin de savoir comment exactement sont stockées ces trois réels (ce pourrait être un tableau par exemple). Par contre il faut évidemment permettre aux utilisateurs de la classe d'accéder à certaines données, on utilise pour cela des méthodes appelées « accesseurs ». Dans une classe « date »,. si les attributs étaient publics, on pourrait y mettre ce qu'on veut. Par contre, si on passe par un accesseur, ce dernier pourra vérifier la validité de la donnée (mois entre 1 et 12 par exemple). On commence en général les noms des accesseurs par « get » ou « set ». Revenons à nos vecteurs. On aimerait permettre à tout le monde de connaître les coordonnées d'un vecteur, mais pas de les modifier individuellement (on laisse cette possibilité aux héritiers). La déclaration de la classe devient donc :

```
class vecteur
 {
  private :  //définition (privée) des attributs
   float x,y,z;
  public :   //les accesseurs en lecture sont publics
   float getx(void) {return x;}
   float gety(void) {return y;}
   float getz(void) {return z;}
  protected :  //seuls les vecteurs et les héritiers ont le droit de
               //modifier une des composantes indépendamment des autres
   void setx(float arg) {x=arg;}
   void sety(float arg) {y=arg;}
```

```
    void setz(float arg) {z=arg;}
  public :    //enfin les méthodes, utilisables (et utiles) par tous
    void set(float a,float b, float c) {x=a,y=b,z=c;}
    void saisir(void);
    void afficher(void)
    float norme(void);
    void multiplier_par(float);
    void additionner(vecteur);
 };
```

Ici les méthodes sont tellement courtes qu'on les définit complètement dans la classe.

10.6) fonctions utilisant des objets, surcharges

Les méthodes vues au dessus s'appliquent à un objet, et un seul. On les appelle par « nomobjet.methode (arguments) ». Mais on peut également créer des fonctions utilisant des objets. Prenons par exemple le produit scalaire : il nécessite deux arguments (des vecteurs) et nous retourne un scalaire (float). Si on ne veut pas utiliser ce produit en l'appliquant spécifiquement à un objet, on créera simplement une fonction :

```
float produit(vecteur v1, vecteur v2) //il n'y a pas écrit vecteur::
 {
  float p=v1.getx()*v2.getx()+v1.gety()*v2.gety()+v1.getz()*v2.getz();
  return p;
 }
```

On est ici à l'extérieur de la classe, nous n'avons donc droit qu'aux membres publics des vecteurs, c'est pourquoi il faut utiliser les accesseurs, car les attributs sont privés.

On appelle cette fonction de manière classique :

```
float x;
vecteur A,B;
x=produit(A,B);
```

On aurait aussi pu définir ce produit comme une méthode d'un vecteur. On aurait alors dû l'appliquer à un vecteur :

```
float vecteur::produit(vecteur v2) //il faut aussi le définir dans la classe
 {
  float p=x*v2.x+y*v2.y+z*v2.z;
  return p;
 }
```

Ici, on l'appelle par `x=A.produit(B)`. La méthode est donc appliquée à A, et reçoit comme argument réel B. Dans la méthode, quand on parle de x c'est x de A, quand on parle de v2.x. c'est l'attribut x de B. Laquelle des deux est la meilleure solution ? Je ne sais pas.

Avez-vous remarqué une bizarrerie ? J'ai déjà défini une méthode nommée produit. Deux méthodes différentes qui ont le même nom, Mr Trau a dû se tromper ! Non, c'est même fait exprès. On peut donner le même nom à deux méthodes si elles ont des « signatures » différentes. On appelle signature le nombre d'arguments, ainsi que leur type. Ici, si j'écris A.produit(x) il appelle la méthode qui attend un float en argument, si j'écris A.produit(B) il sait qu'il doit appeler l'autre. On appelle cela la « surcharge ». Attention, le type retourné ne fait pas partie de la signature, donc on ne peut pas donner le même nom deux méthodes qui n'ont de différent que le type retourné.

On peut surcharger des méthodes, mais aussi des fonctions, et même les opérateurs. Par exemple on peut surcharger l'opérateur *. Pour l'instant, il sait déjà que faire la multiplication entre deux entiers ou flottants

(d'ailleurs, puisqu'il ne fait pas la même chose, cela correspond déjà à de la surcharge). Apprenons lui à multiplier deux vecteurs, mais aussi un vecteur et un flottant. Pour surcharger un opérateur, il faut le précéder dans son entête par le mot « operator ». Les arguments, bien qu'à l'appel ils soient placés des deux côtés du signe *, sont gérés ici de manière classique (le premier en premier, le second en second) :

```
float operator * (vecteur A, vecteur B)
{
 return A.produit(B); //en supposant que j'ai utilisé la deuxième écriture
}
vecteur operator * (vecteur A, float X)
{
 return A.produit(X);
}
vecteur operator * (float X,vecteur A)
{
 return A.produit(X);
}
```

On a donc trois fonctions de même nom (*) avec trois signatures différentes : pour deux vecteurs (retourne un float) et pour un vecteur et un float (retourne un vecteur). Il a fallu lui donner les deux cas, pour le produit avec un float, car le compilateur accepte aussi les fonctions non commutatives.

Par contre je ne peux pas utiliser le même opérateur pour définir le produit vectoriel (il prend aussi deux vecteurs en argument, et bien qu'il retourne un vecteur, il aurait la même signature que le produit scalaire). Mais rien ne m'empêche d'utiliser un autre opérateur, par exemple ^

10.7) constructeur, destructeur, opérateur de copie

Par défaut à chaque fois que l'on crée un objet (on dit aussi « instancie »), le compilateur réserve la quantité de mémoire nécessaire et l'alloue à cet objet. Mais il ne fait rien d'autre. En particulier il n'initialise pas les attributs. On peut vouloir une création d'objet plus sophistiquée (initialisation par défaut, empêcher que certains attributs (surtout les pointeurs) soient vides...). C'est pour cela que vous pouvez réécrire le constructeur. Son entête est

```
nomdelaclasse(liste arguments)
```

On peut remarquer qu'il ne retourne rien, même pas void. Les arguments (s'il y en a) serviront à l'initialisation. Pour le vecteur, il n'y a rien de spécial à faire à l'instanciation, mais je propose néanmoins un constructeur pour initialiser le vecteur :

```
vecteur(float a=0,float b=0,float c=0) {x=a;y=b;z=c;}
```

Le constructeur sera appelé lors de l'instanciation simple d'un objet (`vecteur V1(1,1,2); vecteur V2;`), là où l'on a besoin d'un vecteur constant (`V2=vecteur(1,0,0);`) ou lors d'une création dynamique (`vecteur *p; p=new vecteur;`). Vous pouvez par exemple prévoir un constructeur qui interroge au clavier au cas où des attributs ne sont pas initialisés, qui vérifie que les valeurs initiales proposées sont cohérentes, qui ouvre un fichier pour y chercher les valeurs des attributs,...

Dans certains cas on peut également redéfinir le destructeur (appelé lors de la destruction de l'objet). Son entête est `~nomdelaclasse(void)` (il n'accepte aucun argument). Il ne sera nécessaire que pour désalouer des attributs dynamiques ou fermer des fichiers (ce ne sera pas notre cas).

Dernier point : la copie. Lors d'une affectation (signe =), le compilateur copie tous les attributs, et ça devrait suffire dans tous les cas simples. Dans certains cas (pointeurs, tableaux) où un attribut comporte une référence sur un autre objet, il ne recopie que cette référence, pas l'objet. On peut dans ce cas là redéfinir

l'opérateur =, qui doit obligatoirement être une méthode (pas une fonction). Par exemple si l'on veut pouvoir écrire V=0; (copie d'un entier sur un vecteur, impossible par défaut) :

```
      vecteur operator = (int arg)
        { if(arg==0) x=y=z=0; else cout<<"affectation suspecte\n"; }
```

10.8) exemples complets

Vous pouvez trouver ci-dessous la version complète de cet exemple. Vous pouvez également la télécharger : source C++ ou en pdf.

Je propose également à ceux que ça intéresse une version plus élaborée d'une bibliothèque de calculs vectoriels (pour les usages classiques en mécanique), contenant une classe « point » (deux coordonnées), une classe « vecteur » qui hérite du point, et une classe « torseur » qui hérite du point (pour le point d'application) et contient deux attributs de type vecteur. Cet exemple vous montrera par la pratique l'utilisation de l'héritage dont je n'ai pas parlé ici. Source C++ ou en pdf.

```
#include <iostream.h>

class vecteur
 {
 private :  //la structure interne d'un vrai objet n'a pas à être publique
  float x,y,z;

 protected : //à la rigueur les heritiers peuvent accéder
             //directement aux attributs
  void setx(float a=0) {x=a;}
  void sety(float a=0) {y=a;}
  void setz(float a=0) {z=a;}

 public :
  vecteur(float a=0,float b=0,float c=0) {x=a;y=b;z=c;} //constructeur

//les accesseurs en lecture sont publics
  float getx(void) {return(x);}
  float gety(void) {return(y);}
  float getz(void) {return(z);}

  void affiche(ostream &flux) //flux est en argument car je veux pouvoir utiliser cout
mais aussi tout fichier texte
      {flux<<"["<<x<<","<<y<<","<<z<<"]";}
  void saisie(istream &flux); //si le flux est cin on pose des questions sur cout,
sinon on saisit sans question

  void additionner(float a)
      {x=a+x;y=a+y;z=a+z;}
  void additionner(float a,float b,float c) //ceci est une surcharge.
      {x=a+x;y=b+y;z=c+z;}
  void additionner(vecteur a)               //encore une surcharge.
      {x=x+a.x;y=y+a.y;z=z+a.z;}

  float norme(void) {return(sqrt(x*x+y*y+z*z));}  //calcule la norme
  void normer(void) {float n=norme();x/=n;y/=n;z/=n;} //le modifie (le rend unitaire)

  void multiplier_par(float a) {x=a*x;y=a*y;z=a*z;}
  float prodscal(vecteur v) {return(x*v.x+y*v.y+z*v.z);}

  vecteur operator = (int arg)  //opérateur de copie, prévu ici uniquement pour écrire
V=0
    { if(arg==0) x=y=z=0; else cout<<"affectation suspecte\n"; }


 };  //n'oubliez pas ce ; c'est la fin d'une déclaration
```

```
void vecteur::saisie(istream &f)
 {
  if(f==cin)cout<<"entrez x : ";
  f>>x;
  if(f==cin)cout<<"entrez y : ";
  f>>y;
  if(f==cin)cout<<"entrez z : ";
  f>>z;
 }

//redéfinition des opérateurs (sous forme de fonctions)

ostream& operator << (ostream &f,vecteur v)
 {v.affiche(f);return(f);}

istream& operator >> (istream &f,vecteur &v)
 {
  v.saisie(f);
  return(f);
 }

vecteur operator ^ (vecteur v,vecteur w)     //produit vectoriel
 {vecteur z(
            v.gety()*w.getz()-w.gety()*v.getz() ,
            v.getz()*w.getx()-w.getz()*v.getx() ,
            v.getx()*w.gety()-w.getx()*v.gety()
           );
  return(z);
 }

vecteur operator * (float f,vecteur v)     //produit par un réel
 {vecteur z=v;
  z.multiplier_par(f);
  return(z);
 }
vecteur operator * (vecteur v,float f)     //le prod par un float est commutatif !!!
 {return(f*v);}    //je l'ai déjà défini dans l'autre sens, autant s'en servir !

vecteur operator / (vecteur v,float f)
 {return(v*(1/f));}

float operator * (vecteur v,vecteur w)     //produit scalaire
 {return v.prodscal(w);}

vecteur operator + (vecteur v,vecteur w)     //somme vectorielle
 {vecteur z=v;
  v.additionner(w);
  return(z);
 }

vecteur operator - (vecteur v,vecteur w)     //différence vectorielle
 {return(v+((-1)*w));}

/* petit programme main si l'on veut tester l'objet ***********
int main(void)
 {
  vecteur v(1,1,0),w,z;
  cout<<"la norme de "<<v<<" vaut "<<v.norme()<<"\n";
  cout<<"entrez vos nouvelles coordonnées :\n";
  cin>>w;
  cout<<"la norme de "<<w<<" vaut "<<w.norme()<<"\n";
  cout<<"le prod scal de "<<v<<" et "<<w<<" est "<<v.prodscal(w)
                               <<" (ou "<<v*w<<")\n";
  z=2*(v^w);
  cout<<"le double de leur prod vect vaut "<<z<<"\n";
 }
/* ouf, c'est fini *****************************************/
```

11) pointeurs, tableaux et chaînes

11.1) adresses, pointeurs et références

L'endroit où le compilateur a choisi de mettre une variable ou un objet est appelé **adresse** de la variable (c'est en général un nombre, chaque mémoire d'un ordinateur étant numérotée de 0 à ?). Cette adresse ne nous intéresse que rarement de manière explicite, mais souvent de manière indirecte. Par exemple, dans un tableau, composé d'éléments consécutifs en mémoire, en connaissant son adresse (son début), on retrouve facilement l'adresse des différentes composantes par une simple addition. On appelle **pointeur** une variable dans laquelle on place (mémorise) une adresse de variable (où elle est) plutôt qu'une valeur (ce qu'elle vaut).

On déclare un pointeur par « type_pointé * nom; ». On obtient l'adresse d'un objet par l'opérateur &. Quand à l'opérateur * (dit d'indirection), placé devant un pointeur, il signifie « valeur pointée », j'aime aussi l'appeler « bout de la flèche » (voir schéma ci-dessous).

Voyons cet exemple. La gestion des adresses est simplifiée par rapport à la réalité.

```
int i,j;
int *p,*q;
p=&i;
*p=4;
q=&j;
*q=(*p)+2;
```

Quel est l'intérêt de cet exemple ? Uniquement de vous faire comprendre comment fonctionnent les pointeurs. Nous verrons plus tard des cas où l'on s'en sert (tableaux dynamiques par exemple).

En déclarant int &r; on crée une **référence**. C'est proche d'un pointeur, les deux principales différences sont :

- en parlant de « r » on parle directement de la valeur pointée. Cela simplifie l'écriture du code en évitant toutes ces *. Mais contrairement à ce que pensent les débutants, ce n'est une simplification que si on parfaitement compris le mécanisme des pointeurs (on ne voit plus les pointeurs mais le fonctionnement et les risques restent les mêmes).
- un pointeur est une variable, qui peut au cours du temps comporter des adresses différentes. Une référence pointe toute sa durée de vie sur la même variable (donc r++ est interdit).

En fait, les références sont surtout pratiques pour le passage d'arguments et retour des fonctions (voir le chapitre sur les fonctions), pour le reste ne vous en servez pas dans un premier temps.

11.2) tableaux unidimensionnels statiques

Un tableau est le regroupement en une seule variable de plusieurs éléments (tous les éléments doivent être de même type, mais celui-ci peut être complexe). Avant tout, il faut déclarer le tableau. Le mieux est de déclarer d'abord (de manière globale) comment sera le (ou les) tableau :

```
typedef type_elements type_tableau[nombre_d_elements];
```

Par exemple, si l'on déclare :

```
#define DIM 100
typedef float tableau[DIM];
```

Alors on a défini qu'un tableau sera un regroupement de 100 flottants. Mais on n'a toujours pas réellement créé de tableau, ceci se fait, comme les autres variables, en déclarant (en local si possible) :

```
type_tableau liste de variables;
```

dans notre exemple, pour créer deux tableaux on écrira :

```
tableau x,y;
```

On accède aux éléments d'un tableau par

```
nom_du_tableau[indice]
```

L'indice est un entier qui permet de désigner quel élément précis on désire. C'est obligatoirement un entier, mais ce peut être une constante (x[5]), une variable (x[i]), ou même toute expression qui retourne un entier (x[(2*a+b)/3]). L'indice du premier élément d'un tableau est 0. Pour traiter tous les éléments d'un tableau, il faut :

- expliquer comment traiter le $i^{ième}$
- insérer ce traitement dans une boucle pour traiter tous les i (du premier au dernier).

Par exemple pour mettre à 0 tout le tableau x déclaré précédemment :

```
for(i=0;i<DIM;i++) x[i]=0;
```

L'intérêt principal d'un tableau n'est donc pas de regrouper plusieurs valeurs, mais bien de regrouper des valeurs sur lesquelles on désire faire le même traitement, les mêmes calculs.

Une grosse limitation de ces tableaux est que la dimension (nombre d'éléments réservés lors de la déclaration) doit être connue lors de la compilation, alors qu'on aimerait souvent pouvoir la décider lors de l'exécution. La seule solution est de prévoir, lors de la compilation, une dimension suffisante, et de n'utiliser qu'une partie du tableau lors des exécutions. Par exemple, si je sais que je n'ai jamais plus de 100 étudiants dans une promo, et que je veux mémoriser les notes des GSI2 et des LTM :

```
#define MAX 100
typedef float notes[MAX];
notes gsi2, ltm;
int i, nb_gsi2, nb_ltm;
cout<<"nb de GSI ? ";
cin>>nb_gsi2;
cout<<"nb de LTM ? ";
cin>>nb_ltm;
for(i=0;i<nb_gsi2;i++)
 {
  //traitement de tous les gsi2[i]
 }
for(i=0;i<nb_ltm;i++)
 {
  //traitement de tous les ltm[i]
 }
```

11.3) exemple 1 (objet comportant un tableau)

Pour commencer, voici un exemple de définition d'une classe permettant de d'utiliser des polynômes (exemple : $5x^4+2x^3-x+2$). Un certain nombre de problèmes nécessitent d'utiliser des polynômes. Un polynôme est caractérisé par son degré (ici 5) et ses coefficients (ici 5,2,0,-1,2 si on les donne du poids fort au poids faible). Ce programme définit une classe de polynômes, avec une certain nombre de méthodes qui seront utiles dans tout programme devant traiter des polynômes. Le programme principal donné ici n'est là que pour tester si notre classe est correcte. Vous pouvez voir le code dans une seconde fenêtre, télécharger le source (bouton droit) ou la version pdf.

On se limite aux polynômes de degré 19 maximum. Un polynôme est donc défini par deux attributs : son degré, et un tableau contenant ses coefficients. Comme (presque) toujours pour mes objets, les attributs sont privés. Ils sont accessibles en lecture (get_degre et get_coef), mais pas modifiables directement. Par contre on peut modifier leur valeur, par exemple via les méthodes saisie, copie ou raz. Certaines méthodes n'ont besoin d'aucun argument (par exemple raz) car l'accès aux attributs leur suffit. D'autres ont besoin d'un argument, comme additionner : il faut dire ce que l'on veut additionner au polynôme. On peut remarquer qu'ici j'ai utilisé la surcharge : j'ai donné le même nom à l'addition d'un flottant et à celle d'un autre polynôme.

Comment appelle-t-on cette addition ? Si j'ai déclaré deux polynômes p1 et p2, alors `p1.additionner (3.5);` additionne le flottant à p1, et `p1.additionner(p2);` additionne les deux polynômes, mais seul p1 est modifié (puisque c'est à lui qu'on applique la méthode).

J'ai également surchargé les opérateurs afin qu'ils s'appliquent aux polynômes (là aussi avec surcharges suivant les types d'arguments.

11.4) les chaînes de caractères

En C, comme dans les autres langages, certaines fonctionnalités ont été ajoutées aux tableaux afin de traiter du texte. En C, on représente les chaînes par un tableau de caractères, dont le dernier est un caractère de code nul (\0). Une constante caractères est identifiée par ses délimiteurs, les guillemets " (double quote).

exemples :

```
cout<<"salut";
char mess[]="bonjour"; /* évite de mettre ={'b','o',..,'r','\0'} */
cout<<(mess);
```

mess est un tableau de 8 caractères (\0 compris). On peut au cours du programme modifier le contenu de mess, à condition de ne pas dépasser 8 caractères (mais on peut en mettre moins, le \0 indiquant la fin de la chaîne). On peut également initialiser un pointeur avec une chaîne de caractères :

```
char *strptr="bonjour";
```

Le compilateur crée la chaîne en mémoire, et une variable strptr contenant l'adresse de la chaîne. Le programme pourra donc changer le contenu de strptr (et donc pointer sur une autre chaîne), mais pas changer le contenu de la chaîne initialement créée.

En C++, il y a une classe spécifique pour les chaînes de caractères : les « string ». Ils sont associés à de nombreuses méthodes, bien utiles en cas d'utilisation importante de chaînes de caractères.

Mais déjà en C, la bibliothèque de chaînes (inclure string.h) possède des fonctions utiles à la manipulation

de chaînes :

`int strlen(chaîne)` donne la longueur de la chaîne (\0 non compris)

`char *strcpy(char *destination,char *source)` recopie la source dans la destination, rend un pointeur sur la destination (dest=source quand à lui ne recopie pas la chaîne mais uniquement son adresse !). Faites attention de ne pas dépasser la dimension de la destination, sinon utilisez le suivant.

`char *strncpy(char *destination,char *source,int longmax)` idem strcpy mais s'arrête au \0 ou longmax (qui doit comprendre le \0)

`char *strcat(char *destination,char *source)` recopie la source à la suite de la destination, rend un pointeur sur la destination

`char *strncat(char *destination,char *source,int longmax)` idem, mais plus sûr

`int strcmp(char *str1,char*str2)` rend 0 si str1= =str2, <0 si str1<str2, >0 si str1>str2 (classé alphabétiquement, donc test du premier caractère, si différents test du second, etc...) Les majuscules sont différentes des minuscules (inférieures). Idem pour strncmp, qui lui vérifie la dimension.

Des fonctions similaires, mais pour tous tableaux (sans s'arrêter au \0) sont déclarées dans mem.h. La longueur est à donner en octets (on peut utiliser sizeof) :

```
int memcmp(void *s1,void *s2,int longueur);
void *memcpy(void *dest,void *src,int longueur);
```

Dans ctype.h, on trouve des fonctions utiles (limitées au caractères) :

`int isdigit(int c)` rend un entier non nul si c'est un chiffre ('0' à '9'), 0 sinon. De même : `isalpha` (A à Z et a à z, mais pas les accents), `isalnum` (isalpha||isdigit), `isascii` (0 à 127), `iscntrl` (0 à 31), `islower` (minuscule), `isupper`, `isspace` (blanc, tab, return...), `isxdigit` (0 à 9,A à F,a à f)...

`int toupper(int c)` rend A à Z si c est a à z, rend c sinon (attention, les accents ne sont pas gérés). Egalement `tolower`

11.5) exemple 2 (objets utilisant les chaînes de caractères, tableau d'objets)

Vous pouvez voir le code dans une seconde fenêtre, télécharger le source (bouton droite) ou la version pdf. Cet exemple est volontairement assez court, je n'ai mis qu'un minimum de méthodes pour voir comment ça fonctionne, dans un cas pratique j'aurais bien entendu prévu bien plus de méthodes.

Pour commencer, dans cet exemple, j'ai défini un type « chaîne de 20 caractères » nommé tnom.

Un étudiant possède les attributs nom (de type tnom), et deux notes (TP et exam). Comme à mon habitude, ces attributs sont privés, on ne peut y accéder que par les accesseurs. D'ailleurs, un utilisateur de la classe dispose d'un accesseur à la note moyenne, sans savoir qu'en fait cette note n'est même pas mémorisée dans l'objet mais recalculée à chaque fois. Pour le nom, je n'ai prévu comme accesseur que la saisie au clavier et l'affichage à l'écran.

Une méthode permet de comparer le nom de l'étudiant avec un nom donné en argument (le résultat sera celui de strcmp, donc un entier positif, négatif ou nul). Il serait pratique de surcharger les opérateurs de comparaison = =, <, etc...

Puis dans cet exemple j'ai créé une classe « promo » contenant un tableau d'étudiants (100 au maximum). nb représente le nombre réel d'étudiants, car bien que j'ai réservé de la place pour 100, je peux en utiliser une partie uniquement (mais pas plus). Une fois de plus, à l'extérieur de la classe on ne sait pas comment sont

gérés les attributs : ici, bien que les indices du tableau aillent de 0 à 99, on accède aux étudiants par un numéro entre 1 et 100. Ici aussi on utilise la surcharge : la méthode « affiche » est prévue avec trois signatures différentes : sans arguments, tous les étudiants sont affichés. Avec un argument entier n, seul le $n^{ième}$ étudiant est affiché. Avec un argument de type tnom, l'étudiant portant ce nom est affiché (donc ses notes), du moins s'il existe.

11.6) équivalence d'écriture pointeurs et tableaux

En déclarant, par exemple, `int TAB[10];` l'identificateur TAB correspond en fait à l'adresse du début d'un tableau de 10 entiers. Les deux écritures TAB et &TAB[0] sont équivalentes (ainsi que TAB[0] et *TAB). On définit l'opération d'incrémentation pour les pointeurs par : TAB+1<=>adresse de l'élément suivant du tableau. L'arithmétique des pointeurs en C a cette particularité que l'opération dépend du type de variable pointée, ajouter 1 consistant à ajouter à l'adresse la taille de l'objet pointé. On définit donc l'addition (pointeur+entier) : **TAB+i <=> &TAB[i]**, la soustraction (pointeur - entier), mais également la soustraction (pointeur - pointeur) qui donne un nombre d'éléments. Les opérations de comparaisons entre pointeurs sont donc également possibles.

Déclarons : `int TAB[10],i,*ptr;`

Ceci réserve en mémoire

- la place pour 10 entiers, l'adresse du début de cette zone est TAB,
- la place pour l'entier i,
- la place pour un pointeur d'entier (le type pointé est important pour définir l'addition).

Analysons les instructions suivantes :

```
ptr=TAB; /*met l'adresse du début du tableau dans ptr*/
for(i=0;i<10;i++)
 {
  cout<<"entrez la "<<i+1<<"ième valeur :\n";  // +1 pour commencer à 1
  cin>>*(ptr+i);    // ou TAB[i]
 }
cout<<"affichage du tableau";
for(ptr=TAB;ptr<TAB+10 /* ou &TAB[10] */;ptr++)
  cout<<*ptr;
cout<<"\n";
// attention actuellement ptr pointe derrière le tableau !
ptr-=10; // ou ptr=TAB qui lui n'a pas changé
cout<<*ptr+1;   // affiche (TAB[0])+1
cout<<*(ptr+1); // affiche TAB[1]
cout<<*ptr++;   // affiche TAB[0] puis pointe sur TAB[1]
cout<<(*ptr)++; // affiche TAB[1] puis ajoute 1 à TAB[1]
```

TAB est une "constante pointeur", alors que ptr est une variable (donc TAB++ est impossible). La déclaration d'un tableau réserve la place qui lui est nécessaire, celle d'un pointeur uniquement la place d'une adresse.

Quand on passe un tableau en argument d'une fonction, on ne peut que le passer par adresse (recopier le tableau prendrait de la place et du temps). Voyez cet exemple utilisant les deux écritures équivalentes :

```
void annule_tableau(int *t,int max)
 {
  for(;max>0;max--)*(t++)=0;
 }
void affiche_tableau(int t[], int max)
 {
```

```
  int i;
  for(i=0;i<max;i++) cout<<i<<":"<<t[i])<<"\n";
 }
int main(void)
 {
  int tableau[10];
  annule_tableau(tableau,10);
  affiche_tableau(tableau,10);
 }
```

Il aurait néanmoins été plus simple de définir le type tableau et l'utiliser partout :

```
typedef int tabint[10];

void annule_tableau(tabint t,int max)   //le même bloc d'instructions
void affiche_tableau(tabint t, int max) //le même bloc
int main(void)
 {
  tabint tableau;
  annule_tableau(tableau,10);
  affiche_tableau(tableau,10);
 }
```

11.7) tableaux dynamiques

Les tableaux définis jusqu'ici sont dits « statiques », car il faut qu'au moment de l'écriture du programme le programmeur décide de la taille maximale que pourra atteindre le tableau, sans exagérer pour ne pas consommer trop de mémoire. Mais grâce à l'équivalence d'écriture tableaux-pointeurs, on peut créer des tableaux dont la dimension n'est définie que lors de l'exécution. Exemple :

```
int* tab; //tab est pointeur, il contiendra l'adresse d'un entier, en fait le premier
du tableau.
int nb;
cout<<"combien d'éléments ? "
cin>>nb;
tab=new int[nb]; //ceci réserve de la place pour nb entiers, met l'adresse du premier
dans tab.
```

désormais on peut utiliser tab comme un tableau classique (et écrire tab[i]). On peut même libérer la place prise par le tableau par l'instruction :

```
delete[] tab; //attention, n'oubliez pas les crochets !!
```

Mais si vous ne le faites pas, la mémoire sera de toute façon libérée à la fin du programme.

Les tableaux dynamiques sont donc aussi faciles à utiliser que les tableaux statiques, c'est une caractéristique importante du C++. Ils ne sont néanmoins pas totalement dynamiques : il faut connaître la dimension avant la première utilisation du tableau. Nous ne pourrons donc pas facilement faire un programme qui demande « entrez vos valeurs, terminez par -1 » mais « combien de valeurs ? » puis « entrez vos valeurs ».

11.8) tableaux d'objets, de tableaux...

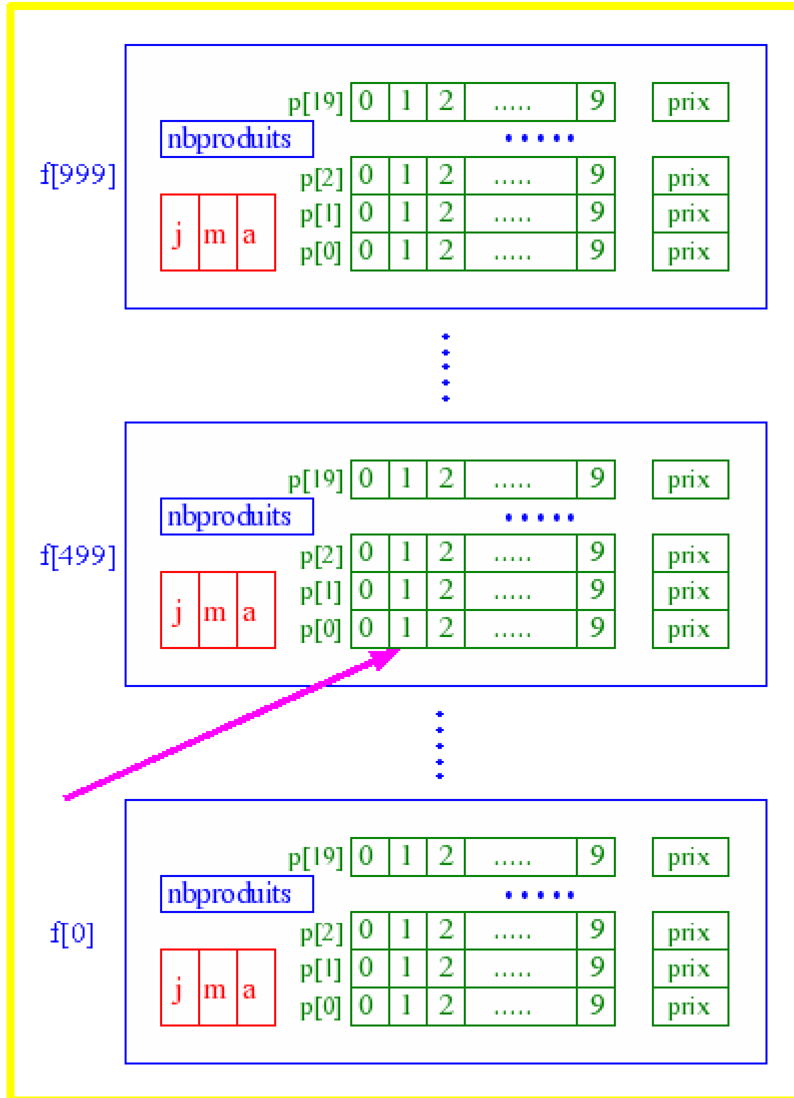
On peut faire un tableau de n'importe quel type d'éléments. Par exemple, voici une matrice 10,10 :

```
#define DIM10
typedef float ligne[DIM]; //une ligne contient 10 float
typedef ligne matrice[DIM]; //une matrice contient 10 lignes
matrice m;
```

```
m[5][2]=0; //on met 0 dans la ligne d'indice 5, colonne d'indice 2.
```

On peut également faire un tableau de pointeurs (et donc de tableaux dynamiques), un tableau d'objets contenant eux-même des tableaux...

autre exemple de déclaration (partielle, il manque des attributs et méthodes) :




```
class date {int j,m,a;};
class produit {char reference[10];
    float prix;};
class facture {date d;int
    nbproduits;produit p[20];};
class compta {facture f[1000];};
    compta livre;
```

livre contient au maximum 1000 factures. `livre.f[499].p[0].reference [1]` correspond au deuxième caractère de la référence du premier produit de la 500ème facture du livre.

11.9) algorithmes pour les tableaux

Supposons maintenant avoir une classe, quelle qu'elle soit, contenant différents attributs, les méthodes saisie, affiche; les accesseurs permettant d'obtenir une valeur (qui servira pour classer les objets). Appelons cette classe « element ». Définissons maintenant une classe « tableau » qui contient un tableau regroupant plusieurs éléments. Nous pouvons alors créer les méthodes saisie et affiche, qui utilisent les méthodes des éléments. Sur le site, vous pouvez voir le code dans une seconde fenêtre, télécharger le source ou la version pdf de cette version de départ. Nous allons maintenant développer des méthodes utiles pour les tableaux (Vous pouvez voir le code, télécharger le source ou la version pdf du programme complet).

Pour commencer, pour **rechercher** un élément d'une valeur donnée, la méthode la plus simple consiste à balayer tout le tableau jusqu'à ce qu'on trouve la valeur désirée (si on arrive à la fin du tableau, cela signifie que la valeur n'est pas présente dans le tableau). Par contre, si le tableau est trié, on peut rechercher par **dichotomie** : regarder si l'élément du milieu est plus grand ou plus petit, pour en déduire s'il faut continuer à chercher dans la première ou la seconde moitié du tableau. Puis recommencer avec le milieu, et ainsi de

suite.

Pour rajouter un élément dans le tableau, si c'est à la fin c'est facile, il suffit de le mettre au bout et d'ajouter 1 à nb (du moins s'il restait de la place). Par contre si c'est à insérer entre les autres, il faut d'abord faire un décalage. Imaginez qu'on ait des voitures dans une suite de garages. Pour en insérer une (sans changer l'ordre), il faut tout décaler : décaler la dernière d'un cran (c'est la seule place de libre actuellement), puis décaler l'avant-dernière, et continuer jusqu'à ce que le box destination soit libre pour y insérer la nouvelle voiture. Pour supprimer un élément, contrairement aux voitures, il n'est pas nécessaire de vider un élément du tableau pour pouvoir l'écraser par un autre (et une fois de plus, il faut tout décaler, puisqu'on les veut continues et dans l'ordre). Pour supprimer la dernière, il suffit de soustraire 1 à nb, même en laissant la valeur puisqu'on ne regarde jamais au delà des nb premières valeurs.

Avant de trier, on commence par prévoir quelques méthodes qui nous serviront. Pour **échanger** deux valeurs, il faut (comme les voitures) : en stocker une dans un endroit temporaire, déplacer l'autre (qui maintenant ne risque plus d'écraser la première), puis ranger la première. Une seconde méthode nous permet de **chercher l'élément le plus petit**, depuis une position donnée du tableau jusqu'à la fin (pour chercher dans tout le tableau il suffit de lui dire de chercher à partir de 0) : on suppose que le premier est le plus petit (pp), puis on teste tous les autres, à chaque fois qu'on en trouve un encore plus petit on remet à jour pp, arrivé au bout pp ne peut être que la position du plus petit. La troisième recherche la position (à partir du début) du **premier élément plus grand** que celui donné en argument (si jamais il n'en trouve pas, il retourne nb, c'est à dire la première position libre).

Pour trier, je vous propose trois algorithmes : le **tri bulle** consiste à comparer chaque élément avec son voisin. Si l'ordre n'est pas bon, on échange les deux. Ce tri est facile à mettre en oeuvre mais est complètement idiot dans la plupart des cas, car une valeur va faire de multiples « sauts de puce » avant d'arriver à sa position définitive. Le **tri sélection** est bien plus malin : chaque échange met une des deux valeurs dans sa position définitive, l'autre par contre étant déplacé moins intelligemment : on cherche le plus petit, on le place en première position (en l'échangeant avec celui qui y était). Puis on ne s'occupe plus que de ceux qui restent, en cherchant le plus petit et le mettant en seconde position, etc... Troisième proposition (pas très efficace dans le cas des tableaux, mais utile dans d'autres cas) : le **tri insertion**. C'est l'algorithme que vous utilisez pour trier vos cartes à jouer : vous comparez la deuxième carte avec la première. Si elles ne sont pas dans le bon ordre, on sort la deuxième et on l'insère au début. Puis on teste la troisième, si elle n'est pas bien placée par rapport aux précédentes, on la sort et on la réinsère à la bonne position. Puis on vérifie la quatrième, etc...

Enfin un petit programme permet de tester ces méthodes.

Table des matières

1) Introduction.......... 1
- 1.1) Organisation - fonctionnement de l'ordinateur.......... 1
- 1.2) Langages de programmation.......... 1

2) Le processus de compilation.......... 2

3) Structure d'un fichier source, définitions.......... 3
- 3.1) identificateurs, séparateurs, commentaires.......... 4
- 3.2) directives du précompilateur.......... 5
- 3.3) structure d'un programme simple.......... 6

4) Les variables simples et opérateurs associés.......... 7
- 4.1) les types entiers.......... 7
- 4.2) Expressions, affectation.......... 7
- 4.3) opérateurs sur les entiers.......... 8
- 4.4) les flottants.......... 8

4.5) le cast....8
4.5) retour sur la structure d'un petit programme....9
5) Les instructions et leur séquencement....10
5.1) définitions....10
5.2) précision sur le « ; »....10
6) Structures de contrôle : les boucles....10
6.1) boucle while (tant que)....10
6.2) boucle do while (faire tant que)....12
6.3) boucle for (pour)....12
7) Structures de contrôle : les branchements conditionnels....13
7.1) If - Else (Si - Sinon)....13
7.2) Switch - Case (brancher - dans le cas)....13
8) Branchements inconditionnels (goto)....14
8.1) goto (aller à)....15
8.2) break (interrompre)....15
8.3) continue (continuer)....15
8.4) return (retourner)....15
8.5) exit (sortir)....16
9) Fonctions....16
9.1) Définition générale....16
9.2) procédure (fonction ne retournant rien)....16
9.3) fonction retournant une valeur....17
9.4) passage d'arguments (par valeur, par référence,...)....18
9.5) appel de fonctions, prototypes....19
9.6) Exemple....19
9.7) Récursivité, gestion de la pile....20
10) Objets....21
10.1) introduction....21
10.2) la classe....21
10.3) définition des méthodes....21
10.4) utilisation d'un objet....22
10.5) accessibilité aux membres d'une classe....23
10.6) fonctions utilisant des objets, surcharges....24
10.7) constructeur, destructeur, opérateur de copie....25
10.8) exemples complets....26
11) pointeurs, tableaux et chaînes....28
11.1) adresses, pointeurs et références....28
11.2) tableaux unidimensionnels statiques....29
11.3) exemple 1 (objet comportant un tableau)....30
11.4) les chaînes de caractères....30
11.5) exemple 2 (objets utilisant les chaînes de caractères, tableau d'objets)....31
11.6) équivalence d'écriture pointeurs et tableaux....32
11.7) tableaux dynamiques....33
11.8) tableaux d'objets, de tableaux....34
11.9) algorithmes pour les tableaux....34