La Syntaxe du C++

Licence de Physique

Contents

1 Affichage à l'écran et lecture du clavier

La gestion des entrée/sortie avec les périphériques standards que sont le clavier et l'écran se fond, en C++, au moyen de 2 classes, ***istream*** et ***ostream***. Ces classes sont définies dans le fichier **iostream**, qu'il faut inclure au début du programme. Nous reviendrons en détail plus tard sur ce qu'est une classe ; mais le but de cette partie est juste de vous monter comment on peut lire le clavier et écrire sur l'écran.

1.1 Affichage

```
#include <iostream>
using namespace std;
int main() // entête du programme principal
{
  // début
  int a,b;  // déclaration des objets a et b de la classe int
  a=1;      // affectation: a prend la valeur 1
  b=a+1;    // affectation b prend la valeur 2
  int c;    // déclaration de la objet c de classe int
  c=a+b;    // affectation: c prend la valeur a+b c'est à dire 3
  cout<<"la somme de "<<a<<" et "<<b<< " = "<<c<<endl;// affichage à l'écran.
  return 0;// fin
}
```

1.1.1 Remarques

- Si en C la fonction *main()* est de type ***void***, en C++ elle est de type ***int*** ; une fonction *main()* de type ***void*** peut selon les compilateurs, provoquer ou non une erreur. Par contre le **return 0;** n'est pas obligatoire dans le *main()*.
- Le programme va afficher:

```
la somme de 1 et 2 = 3
```

- Les caractères entre `" "` sont considérés comme une chaîne de caractères et écrits tels quels à l'écran. Par contre les caractères `a,b,c` ne sont pas entre `" "`. Cela signifie que ce sont des noms d'objets, et qu'il faut afficher le contenu de ces objets (et non pas leur nom).
- Le terme ***cout*** symbolise l'écran. Les signes `<<` sont évocateurs: on envoie ce qui suit vers l'écran. ***endl*** signifie "on vide le buffer et on va à la ligne ("end line")".

1.1.2 Remarques plus techniques

1. Jusqu'à il y a peu de temps, il existait un certains nombre de classes prédéfinies qui portaient le même nom, faisaient plus ou moins la même chose mais rendait la portabilité des programmes hasardeuses ; il a été décidé de standardiser ces classes en utilisant la STL (Standard Template Library). Afin de preciser qu'on utilise des

classes de la STL, les fichiers inclus (#include <name.h>) non plus l'extension ".h" (donc #include <name>) et on ajoute ***using namespace std;***

2. `iostream` est un fichier déjà présent sur l'ordinateur. Ce fichier contient des informations sur des commandes C++ d'affichage à l'écran et de saisie de touches appuyées au clavier. Iostream vient de l'anglais: Input (entrée), Output (sortie), Stream (flux d'information).

3. Dans la commande **#include**, si le fichier à inclure est entre < >, cela signifie que le compilateur va chercher ce fichier dans un répertoire particulier, prédéfini. On peut inclure des fichiers en remplaçant les < > par des " " : si aucun chemin n'est spécifié entre les guillemets, alors le fichier doit se trouver dans le répertoire courant.

4. ***cout*** est **un objet** C++ de la **classe** *ostream*. Cet objet est associé à l'écran comme expliqué ci-dessus. Le signe << est un **opérateur** de cette classe à qui on a donné le sens précis d'afficher à l'écran. Tout cela est contenu dans le fichier *iostream* qui est lui même un programme C++. L'existence de cette classe simplifie donc la programmation pour les suivants. C'est l'esprit du C++. Vous apprendrez à écrire vous même des classes et des opérateurs par la suite.

1.2 Lire le clavier

Il peut être intéressant que l'utilisateur puisse lui-même entrer les valeurs de a et b. Pour cela l'ordinateur doit attendre que l'utilisateur entre les données au clavier et les valide par la touche *entrée*.

```
# include <iostream>
using namespace std;
int main()     // entête du programme principal
{
  // début
  int a,b;                    // déclaration des objets a et b de la classe int
  cout<<"Quelle est la valeur de a ?" <<flush;
  cin>>a;                   // lire a au clavier, attendre return
  cout<<"Quelle est la valeur de b?" <<flush;
  cin>>b;                  // entrer b au clavier puis return
  int c;                     // déclaration de la objet c de la classe int
  c=a+b;                  // affectation: c prend la valeur a+b
  cout<<"la somme de "<<a <<" et "<< b << " vaut "<<c<<endl ;  // affichage à l'écran.
  return 0;// fin
}
```

1.2.1 Remarques

- Cette fois-ci vous avez compris que l'objet ***cin*** appartient à la classe ***istream*** et représente le clavier. Le signe >> est un opérateur associé à cet objet et qui a pour effet de transférer les données tapées au clavier dans l'objet qui suit (une fois la touche *entrée* enfoncée).

- L'instruction ***flush*** à la fin de la phrase d'affichage à pour effet de d'afficher la phrase à l'écran sans faire de retour à la ligne. Si on ne met rien (ni ***flush***, ni ***endl***) la phrase n'apparaît pas tout de suite à l'écran.

2 Déclaration et affectation des objets

2.1 Les déclarations d'objets de base

Pour stocker une information dans un programme, on utilise un objet qui est symbolisée par une lettre ou un mot. (Comme `a,b,c` précédemment). On choisit la **classe** de cet objet, selon la nature de l'information que l'on veut stocker (nombre entier, ou nombre à virgule, nombre complexe, ou série de lettres, ou matrice,...).

Voici les différentes classes de base qui existent en C++:

Déclaration: `classe objet;`	signification	Limites
`int a;`	entier	voir remarque 1
`float c;`	réel	$\pm 10^{\pm 38}$ à 10^{-7} près
`double d;`	réel double précision	$\pm 10^{\pm 308}$ à 10^{-13} près
`char e;`	caractère	256 caractères
`char *f;`	chaîne de caractère	

Remarques:

1. Les limites des objets dépendent du nombre d'octets (ensemble de 8 bits) utilisés pour leur stockage. Les caractères sont stockés sur 1 octet, soit $2^8 = 256$ caractères au total. Pour les entiers, cela dépend des systèmes d'exploitation ; sur certains (DOS,Windows) ils sont stockés sur 2 octets (limites=-2^{15} à $2^{15}-1$), sur d'autres (HP, linux) sur 4 octets (limites=-2^{31} à $2^{31}-1$) et même parfois sur 8 octets (DEC).

2. Quand utiliser ***float*** plutôt que ***double*** ? Puisque ce dernier est de toutes façons plus précis. Réponse: Stocker un objet de la classe ***double*** dans la mémoire de l'ordinateur (ou dans un fichier) prends plus de place. Par contre l'ordinateur calcule aussi vite avec ***float*** qu'avec ***double***, car les processeurs actuels sont structurés pour cela. Donc si économiser de la place mémoire est crucial pour votre programme, et que la précision n'est pas nécessaire, (mais cela est rare) il est préférable d'utiliser ***float***, sinon utilisez (en général) ***double***. **Ayez le réflexe "*double*"**...

3. Les classes ci-dessus sont les classes de base standard en C++ qui existent en C. Dans le vocabulaire du C, on dirait plutôt **type** à la place de **classe**, et **variable** à la place de **objet**. Par exemple en écrivant `double d`; on dirait que `d` est une variable du type `double`.

4. Vous pouvez utiliser d'autres classes plus élaborés (qui ne sont plus des classes de base), comme les entiers à précision illimitée, les complexes, les vecteurs, les matrices,.. à condition cette fois ci d'inclure le fichier "`.h`" approprié au début du programme.

2.2 Portée des objets

Il existe 2 grandes catégories d'objets, **les objets globaux et les objets locaux**. Les objets **globaux** se déclarent juste après les includes. Ils sont connus dans tous les blocs qui les suivent. Ils sont, comme nous l'avons déjà dit, **à éviter**. Les objets **locaux ne vivent que dans le bloc** (`{...}`) dans lequel ils ont été déclarés ; ils sont détruits dès la sortie du bloc.

2.3 Initialisation d'un objet de base

On peut déclarer un objet et l'initialiser en même temps, de différentes manières :

```
int i=0;
float Pi=3.14;
double pi=3.1415,x1=3.15e1;
double y=x1;
char mon_caractere_prefere = 'X';
char *texte="que dire de plus?";
```

`i` est un ***int*** qui vaut **0**, `Pi` un ***float*** qui vaut **3.14**, `pi` est un ***double*** qui vaut **3.1415**, x1 est un ***double*** qui vaut **31.5**, etc...

Remarques

1. Dans le nom des objets, le **langage C++ fait la différence entre les majuscules et les minuscules**. On peut choisir tous les caractères de l'alphabet, utiliser des chiffres et le caractère "_". Cependant **un nom ne peut commencer par un chiffre et *ne peut contenir de caractères spéciaux*** (lettre accentuée, opérateurs, caractères de ponctuation, ...).
2. On peut initialiser un objet avec un objet déjà existant comme `y=x1.`
3. Un caractère est entre le signe *quote* " ' ", comme `'X'.` Un texte (chaîne de caractères) est entre guillemets " " ".
4. On pourra bien sûr modifier ces valeurs dans la suite du programme.

2.4 Complément : Précision à l'affichage

(il faut inclure *<iomanip>*)

```
double c=3.14159;
cout<<setprecision(3)<< c << endl;
```

Affichera: 3.14

2.5 Conversions de classe

On peut affecter un objet à partir d'un objet d'une autre classe, **lorsque cela a un sens** (i.e. lorsque ça a été prédéfini) . On parle de conversion ou **cast**. Ces cast peuvent avoir lieu sur des objets ou des pointeurs (la conversion d'un pointeur d'une classe en un pointeur d'une autre classe est très utilisée).

```
int i;
float f=3.14,g;
i= int(f);                      // convertion de float à int. i contiendra 3.
cout<<i<<endl;
g=float(i);                   // conversion de int à float.
cout<<int('A');           // Conversion de char à int. Affichera 65 à l'écran qui est le code ASCII de A
cout<<char(65)<<endl; //Conversion de int à char. Affichera A à l'écran qui est le caractère qui a le code 65.
```

2.5.1 Remarque Importante

Certaines conversions peuvent être faites de façon implicite, c'est-à-dire sans spécifier la classe. Cependant ceci ne peut se faire sans risque dans n'importe quelle situation. Essayer le programme ci-dessous :

```
#include <iostream>
using namespace std;
int main()
{
  float f=3.14,f1,f2;
  int i=1,j,k,l;
  char c='B',c1,c2;
  f1=i/2*1.0;
  f2=1.0*i/2;
  cout<<"f1=f2?"<<endl;
  cout<<f1<<" "<<f2<<endl;
  j=f;
  cout<<"Conversion implicite de float->int"<<endl;
  cout<<j<<endl;
  k=c;
```



```
  cout<<"Conversion implicite de char->int"<<endl;
  cout <<k<<endl;
  l=65;
  c1=l;
  c2=l+256;
  cout<<"Conversion implicite de int->char"<<endl;
  cout <<c1<<" "<<c2<<endl;
}
```

1. Comme vous l'avez constaté, le plus gros danger vient de la conversion **int->float** (il en serait de même pour `int->double`). En effet *f1* et *f2* sont mathématiquement équivalent mais le résultat numérique est différent. Pourquoi ? Cela vient de la façon dont le compilateur prépare les opérations. Dans *f1=i/2*1.0*, il code la première opération dans un entier (car *i* et *2* sont des entiers) ; le résultat de la deuxième opération est codée dans un ***float*** car *1.0* est un ***float***. Pour *f2*, la première opération est codée dans un ***float*** car *1.0* est un ***float*** ; il en est de même pour la seconde. Pour éviter ce genre de problème on prendra l'habitude d'écrire *f1*=**float**(*i*)/2*1.0.
2. La ligne *j=f* génère à la compilation un message du type "*warning: assignment to 'int' from 'float*"' qui vous indique que cette conversion implicite provoquera une perte d'information. Ce message vous informe d'une éventuelle erreur dans votre programme.
3. Enfin, la conversion de **int->char** donne le même résultat pour *65* et *321=65+256*. Souvenez vous qu'un caractère n'est codé que sur 1 octet, donc *255+1=0*, *255+2=1*, ...

3 Les instructions de base

3.1 Les Conditions

Une condition est quelque chose qui est *vrai* ou *faux*. En C++, comme en C, *faux* est synonyme de **0** et toutes les autres valeurs numériques sont vraies. Souvent, pour des raisons de lisibilité, on utilise **1** pour *vrai*. Voici la syntaxe générale qui permet d'obtenir une expression vraie ou fausse.

3.1.1 Les opérateurs de comparaison:

Signification	symbole
supérieur à	>
inférieur à	<
supérieur ou égal à	>=
inférieur ou égal à	<=
égal à	==
différent de	!=

Attention à la confusion possible: *a==2* sert à comparer l'objet *a* avec *2* (et ne change pas la valeur de *a*). Par contre *a=2* met la valeur *2* dans l'objet *a*.

3.1.2 Les opérateurs logiques

Signification	symbole
et logique	&&
ou logique	||
non logique	!

Remarque Lorsqu'une condition est évaluée comme $i \leq 30$, la valeur rendue est de classe entier (**int**). Elle est différente de *0* si la condition est vraie et *0* sinon.

3.2 Les Boucles

3.2.1 for(initialisation ; condition ; incrémentation) {instructions}

La syntaxe de la boucle *for* dans l'exemple ci-dessous signifie : Initialise i à 10 ; si $i \leq 30$, exécute les instructions du bloc {...} qui suit le *for*, ajoute 2 à i et si $i \leq 30$ recommence.

```
#include<iostream>
using namespace std;
int main( )
{
  int i;
  int j;
  for(i=10;i<=30;i=i+2)
  {
    j=i*i;
    cout<<i<<"\t"<<j<<endl; // le caractère \t est une tabulation
  }
}
```

Ce programme produira:

10 100
12 144
14 196
etc...
30 900

3.2.2 do { instructions} while (condition);

signifie : Fait le bloc d'instructions tant que la condition est vraie.

```
#include<iostream>
using namespace std;
int main( )
{
  int MAX=11;
  int i=1,j;
  do
  {
    j=i*i;
    cout<<i<<"\t"<<j<<endl;
    i=i+1; // Ne pas oublier l'incrémentation
  }
  while(i<MAX); // i < MAX est la condition
}
```

produira:

1 1
2 4
etc...
10 100

3.2.3 while (condition) { instructions }

signifie : Tant que la condition est vraie fait le bloc d'instructions.

```
#include <iostream>
using namespace std;
int main ( )
{
  int MAX=11;
  int i=1, j;
  while(i<MAX) // condition
  {
     j=i*i;
     cout<<i<<"\t"<<j<<endl;
     i=i+1; // ne pas oublier l'incrémentation
   }
}
```

Produira:

1 1
2 4
etc...
10 100

Remarque sur les boucles: On peut forcer la sortie d'une boucle avant que la condition de fin soit réalisée en utilisant la commande ***break***. Nous en verrons un exemple au paragraphe suivant

3.3 if (condition) {instructions} else {instructions}

```
#include<iostream>
using namespace std;
int main()
{
   int i=5,j;
   char test='N';
   while(test!='O') // tant que la valeur de test est différente de 'O'
   {
      cout<<"entrer un nombre entre 1 et 10 "<<endl;
      cin>>j;
      if(j>10)
       {
          cout<<"Comme tu ne sais pas lire je refuse de jouer avec toi"<<endl;
          break; //on sort du while
       }
       if(i==j)
       {
           cout<<"Bravo vous avez trouvé le nombre mystérieux!"<<endl;
           test='O';
       }
       else
         cout<<"Non ce n'est pas le bon nombre"<<endl;
   } // fin du while
} // du programme
```

3.3.1 Exercice : Résolution d'une équation du second degré

Ecrire un petit programme qui

- demande en entrée 3 réels a, b et c
- affiche le discriminant de l'équation $ax^2 + bx + c = 0$
- Donne la (les) solution(s) réelle(s) si elle(s) existe(nt)

3.3.2 Complément : Exercice sur les nombres aléatoires

Faire un programme qui au départ choisit un nombre au hasard entre 0 et 1000 (se servir de la section suivante), puis demande à l'utilisateur de le trouver, en répondant "trop grand" ou "trop petit" à chaque essai. L'utilisateur a droit a un nombre limité d'essais.

Pour générer un nombre entier p aléatoire, entre 0 et N inclus. Il faut ces 2 fichiers:

```
#include <ctime>
#include <cstdlib>
```

Puis dans le programme, l'instruction suivante ne doit apparaître qu'une seule fois; elle permet d'initialiser les tirages à partir de la date.

```
srand(time(NULL));
```

Puis au moment de choisir un nombre au hasard:

```
p=rand()%(N+1);
```

Explications: ***rand()*** génère un nombre entier aléatoire entre 0 et MAX_INT (le plus grand entier). Ensuite **%** signifie modulo, ***a%b*** est le reste de la division de ***a*** par ***b***. Ainsi ***rand()%(N+1)*** est le reste de la division du nombre choisit par `rand()` par ***N+1***. C'est donc un entier compris entre *0* et *N* (inclus), ce que l'on veut.

3.4 switch

L'instruction **switch** est un peu analogue au **if** ; elle peut être remplacée par une succession de **if...else...** mais elle présente l'avantage d'être plus légère à écrire et à relire. La syntaxe générale est de la forme :

```
switch(var)
{
       case value1: instruction1; break;
       case value2: instruction2; break;
       case value3: instruction3;
       case value4: instruction4; break;
       default: instruction_defaut; break;
}
```

où *var* est soit de type **int** soit de type **char** et *valueX* est du type de *var*. Selon la valeur de *var*, faire telles ou telles instructions. Dans l'exemple précédant, si *var=value1* on ne fera que les *instruction1* ; idem pour *value2* et *value4*. Par contre si *var=value3* on effectue *instruction3* et *instruction4* car il n'y a pas de **break**. Enfin si *var* n'est égale à aucune des *valueX*, on fait *instruction_ defaut*. Cette dernière ligne est facultative.

4 Les tableaux

Un tableau est suite d'objets de même la classe.

4.1 déclaration

La façon de déclarer un tableau est la suivante:

```
int tab1[100]; //tab1 est un tableau de 100 cases contenant chacune une objet de la classe entier
float tab2[20]; //tab2 est un tableau de 20 cases contenant chacune une objet de classe réel
char tab3[10]; //tab3 est un tableau de 10 cases contant chacune une objet de classe char. On dit aussi que c'est u
```

Attention: la taille du tableau doit être un nombre *constant* ; ça ne peut pas être la valeur d'un objet (**int**). Si l'on veut créer un tableau de taille variable, voir la section 5pour plus d'information.

4.2 Affectation

Pour mettre la valeur 3 dans la case numéro 0, on écrit naturellement:

```
tab1[0]=3;
```

Attention:

si on déclare `int tab1[N]`, **les N cases sont numérotées de 0 à N-1** . Si on se trompe, si on écrit dans une case hors de limites, cela peut être la cause du plantage de votre programme. Sachez que la plupart des bugs que vous risquez de créer proviendront de ce problème.

4.2.1 Exercice

Complétez les signes "?" dans le programme suivant

```
//pour calculer le produit scalaire de deux vecteurs
#include<iostream>
using namespace std;
const int dim=3; //declaration d'une constante de type entier
int main()
{
  double v1[dim], v2[dim],ps;
  int i;
  v1[?]=1; v1[?]=2; v1[?]=3;
  v2[?]=3; v2[?]=-2; v2[?]=1;
  ps=0.;
  for (i= ? ;i< ? ;i=i+1)
      ps=ps+v1[i]*v2[i];
   cout<<"le produit scalaire est:"<<ps<<endl ;
}
```

Remarques

- Dans le programme ci-dessus, on a déclaré une ***constante dim***. Cela permet de changer facilement la taille des tableaux: il suffit de changer la valeur de ***dim***.
- On peut affecter un tableau au moment de sa déclaration ; par exemple

```
double v1[3]={1.,2.,3.}
```

5 Les pointeurs

La notion de pointeur est importante dans le langage C et C++. Elle est réputée comme étant difficile et technique; nous espérons que vous aurez néanmoins les idées claires après la lecture de cette section.
Nous introduisons rapidement la notion de pointeur, et montrons comme exemple, son intérêt pour créer des **tableaux de taille variable** au cours du programme.

5.1 Déclaration et affectation d'un pointeur

Rappelons déjà ce qu'est un objet. Par exemple:

```
double i;
```

Cette instruction a pour effet de réserver une "case" en mémoire de l'ordinateur, permettant de stocker un nombre réél. Cette case s'appelle i.

Bien sûr, cette case se trouve quelque part dans la mémoire de l'ordinateur. Elle a un certain emplacement, caractérisée par son adresse, appelée **pointeur**.

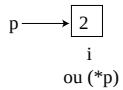
Il faut donc retenir que **pointeur d'un objet** signifie **adresse d'un objet** dans la mémoire de l'ordinateur.

On peut avoir accès à l'adresse de la "case" i en faisant:

```
double i=2.;  // déclare l'objet i et affecte la valeur 2.
double *p;    // déclaration du pointeur p sur un double
p=&i;      // p devient l'adresse de i
```

Grâce au signe *, la deuxième ligne déclare p comme étant un pointeur sur un **double** (désignant une case mémoire qui est sensée contenir un réel). Remarquons que p est en fait un objet de type ***double****. On peut écrire **double*** p ou **double** ***p** ; cependant par convention, on utilise plutôt le seconde notation.

Le signe `&i` signifie l'adresse de l'objet i. Donc la troisième ligne met dans p l'adresse de l'objet i.



Voici une représentation de la mémoire de l'ordinateur:

adresse des cases	contenu de la case	nom de variable
10000	...	
10004	10012	p
10008	...	
10012	2.	i
10016	...	

Pour afficher le contenu de l'objet i on a maintenant deux possibilités:

```
cout<<i;
```

ou:

```
cout<<(*p);
```

qui est équivalent car **(*p) signifie le contenu de la "case" où pointe p.**

```
cout<<p;
```

donne l'adresse pointée par p (i.e. celle de i).

Pour modifier le contenu de l'objet i on a maintenant deux possibilités:

```
i=5.3;
```

ou:

```
(*p)=5.3;
```

qui est équivalent.

Attention: avant d'effectuer l'opération d'écriture: `(*p)=5.3` ; il faut être sur que l'adresse du pointeur correspond à une "case" existante. Vous avez donc compris que avant d'écrire, il faut prendre le soin de réserver de la place mémoire.

5.1.1 Les Tableaux statiques

En fait vous avez déjà utilisé des pointeurs... En effet lors de la déclaration du tableau *tab*

```
float tab[6];
```

vous réservez 6 cases mémoires de type **float**. Comme la dimension de ces tableaux est forcément constante, on parle de *tableau statique*. La variable *tab* n'est rien d'autre qu'un *pointeur sur un float*. Ainsi l'instruction

```
tab[0]=2.3 est parfaitement équivalente à *tab=2.3
tab[1]=2.5 est parfaitement équivalente à *(tab+1)=2.5
...
```

Ceci explique pourquoi les tableaux ont leurs cases numérotées de 0 à N-1 :*(tab+i) signifie "je saute i cases" (dont la taille dépend du type du pointeur).

Il est possible de déclarer des tableaux de plusieurs dimensions :

```
double tab[10][10]
```

L'élément *tab[i][j]* correspond par exemple à la ligne i et la colonne j d'une matrice.

5.2 Allocation dynamique de la mémoire

Il est possible de réserver n cases mémoires, par l'instruction:

```
int n;
cout<<"entrez n"<<flush;
cin>>n;
float *pValeur; //on déclare le pointeur pValeur
pValeur=new float[n]; // on réserve n cases mémoire de la classe float
```

Cela s'appelle une **allocation dynamique** de la mémoire, car elle est faite au cours de l'exécution du programme et non par le compilateur comme dans les tableaux statiques. C'est la commande ***new*** qui permet de faire ceci.

A près cela, *pValeur* **pointe sur la première case réservée :** on peut légitimement écrire dans cette première case par l'instruction:

```
(*pValeur)=1;
```

ou (ce qui est équivalent)

```
pValeur[0]=1;
```

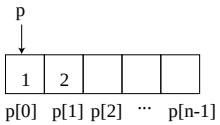
On peut de même écrire dans la case suivante par:

```
*(pValeur+1)=2;
```

ou

```
pValeur[1]=2;
```

etc... jusqu'à pValeur[n-1].



A la fin de l'utilisation, n'oubliez pas de **libérer l'emplacement mémoire** par l'instruction:

```
delete [] pValeur;
```

Remarques :

1. Pour chaque `new` il faut un `delete`
2. Pour libérer la mémoire d'un tableau dynamique il faut faire `delete [] tab`
3. Si l'on veut ne réserver qu'une seule case mémoire au lieu d'un tableau, il suffit de faire:

```
int *p;
p=new int(); // on réserve une case mémoire de classe int.
*p=1;
delete p;   // pour libérer la place mémoire
```

4. Un tableau de caractères est aussi appelé une **chaîne de caractères**. On utilise souvent la déclaration:

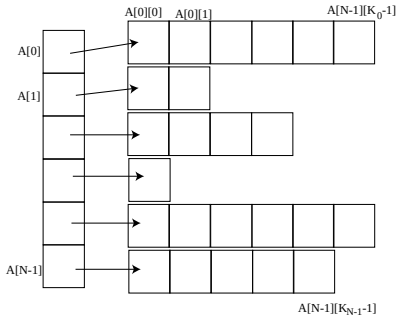
```
char *chaine="toto";
```

   qui déclare *chaine* comme étant un pointeur sur un caractère, pouvant pointer sur les 4 caractères "toto".

5. Il est possible de s'allouer dynamiquement des tableaux de plusieurs dimensions ; par exemple, pour une matrice de **float**

```
float **A; //Ceci est un pointeur sur un pointeur de float
A= new float*[10]; //on crée 10 pointeurs de float
for (int i=0; i<10;i++) A[i]=new float[10]; //on crée 10 cases de float pour chaque A[i]
```

   L'avantage de cette méthode est que votre matrice peut avoir un nombre de colonnes différents pour chaque ligne.



Exercice Modifier le programme de la leçon "tableaux" sur le produit scalaire de deux vecteurs, en créant maintenant ces deux vecteurs de façon dynamique: le programme demande à l'utilisateur la taille n des vecteurs, et les composantes des vecteurs.

6 Les fonctions

Une fonction est un petit sous programme qui utilise (ou pas) certains paramètres que lui passe la fonction qui l'appelle (cela peut être la fonction principale ou *programme principal* ("main")). Une fonction peut modifier un objet, remplir un tableau, écrire une information à l'écran...

Une fonction peut renvoyer **une (et une seule) valeur** d'une certaine classe ou ne pas renvoyer de valeur du tout.

6.1 Fonction sans objet de retour

on met `void` (vide en anglais).

```
# include <iostream>
using namespace std;
//========declaration de la fonction carre=============
void carre(int i)
{
   int j; // declaration d'un objet local
   j=i*i;
   cout<<"le carré de "<<i<<" est "<<j<<endl;
}
//====declaration de la fonction principale =============
int main()
{
   int x;
   cout<<" entrer x "<<endl;
   cin>>x;
   carre(x); // appel de la fonction carre
   return 0;
}
```

Remarques:

1. Dans le bloc `{...}` de la fonction *carre* on a déclaré l'objet j.Par conséquent, **cet objet n'est connu que dans ce bloc et pas ailleurs.** On dit que c'est un objet **local**. On ne peut pas l'utiliser dans la fonction *main*. De même l'objet x déclaré dans le bloc de la fonction *main* n'est pas connu ailleurs: On ne peut pas l'utiliser dans la fonction *carre*.
2. La fonction *main* appelle la fonction *carre* et lui passe un paramètre qui est l'objet x choisi par l'utilisateur.
3. La déclaration de la fonction *carre* montre que cette fonction prend un paramètre (un objet de la classe **int**) et ne renvoie rien (à cause du préfixe **void**). De même la fonction *main* ne prend aucun paramètre, et renvoie 0, un entier (*main()* est de type **int**) .

6.2 Fonction avec objet de retour

```
#include<iostream>
using namespace std;
//======declaration de la fonction carre =================
int carre(int i)
```

```
{
     int j;
     j=i*i;
     return j; //on renvoie le résultat au programme principal
}
//===declaration de la fonction principale ===============
int main()
{
     int x;
     cout<<"entrer x "<<endl;
     cin>>x;
     int y;
     y=carre(x); // appel de la fonction carre
     cout<<"Le carré de "<<x<<" est "<<y<<endl;
     return 0;
}
```

Remarques

La seule différence avec l'exemple précédent est que la fonction *carre* renvoie son résultat, par un objet de la classe **int**. Pour cela on utilise l'instruction ***return***. Et pour appelé cette fonction, on utilise une la syntaxe *y=carre(x)* si bien que le résultat est tout de suite stocké dans l'objet *y*.

6.3 Transmission des arguments d'une fonction.

Il existe trois façons de transmettre un argument à une fonction : par recopie, par pointeur et par référence.

6.3.1 Transmission par recopie

Ce mode de transmission est le mode par défaut ; il est parfois appelé transmission par valeur.

```
#include<iostream>
using namespace std;
//declaration de Echange (ne renvoie rien car void)
void Echange(int a, int b)
{
        int c;
        cout<<"Echange: Avant permutation n="<<a<<" p="<<b<<endl;
        c=a; a=b; b=c;
        cout<<"Echange: Apres permutation n="<<a<<" p="<<b<<endl;
}
int main()
{
        int n=10, p=20;
        cout<<"main:  Avant appel a Echange n="<<n<<" p="<<p<<endl;
        Echange(n,p);
        cout<<"main:  Apres appel a Echange n="<<n<<" p="<<p<<endl;
}
```

6.3.2 Exercice

Exécuter ce programme ; que se passe-t-il ?

Remarques :

- Noter que les noms *a* et *b* dans *Echange* sont arbitraires ; ils auraient pu s'appeler *n* et *p* par exemple sans, pour autant, avoir le moindre lien avec *n* et *p* de la fonction *main()*.
- Bien que dans la fonction *Echange* la permutation ait été faite, dans le *main()*, rien n'a changé. Cela est du au fait que les valeur de *n* et *p* sont recopiées dans les objets locaux *a* et *b* de la fonction *Echange*. Ces objets sont effectivement permutés mais ils sont détruits en sortant de *Echange*. Pour cet exemple, ce n'est pas la bonne façon de procéder.

6.3.3 Transmission par référence

Ce mode de transmission est propre au C++.

```
#include<iostream>
using namespace std;
//declaration de Echange (ne renvoie rien car void)
void Echange(int &a, int &b)
{
        int c;
        cout<<"Echange: Avant permutation n="<<a<<" p="<<b<<endl;
        c=a; a=b; b=c;
        cout<<"Echange: Apres permutation n="<<a<<" p="<<b<<endl;
}
int main()
{
        int n=10, p=20;
        cout<<"main:  Avant appel a Echange n="<<n<<" p="<<p<<endl;
        Echange(n,p);
        cout<<"main:  Apres appel a Echange n="<<n<<" p="<<p<<endl;
}
```

6.3.4 Exercice

Exécuter ce programme ; que se passe-t-il ?

Il fonctionne donc correctement. La seule différence par rapport à la transmission par recopie est l'apparition de "&" devant les paramètres à la déclaration de la fonction *Echange* signifiant que l'on passe la valeur et l'adresse de l'argument ; il n'y a plus recopie. **Cette méthode est donc celle que nous utiliserons à chaque fois que nous aurons besoin de modifier un argument.**

6.3.5 Complément : Transmission par pointeur

Vous pouvez sauter cette section dans un premier temps.

```
void Echange(int *a, int *b)
{
        int c;
        cout<<"Echange: Avant permutation n="<<*a<<" p="<<*b<<endl;
        c=*a; *a=*b; *b=c;
        cout<<"Echange: Apres permutation n="<<*a<<" p="<<*b<<endl;
}
int main()
{
        int n=10, p=20;
        cout<<"main:  Avant appel a Echange n="<<n<<" p="<<p<<endl;
```

```
        Echange(&n,&p);
        cout<<"main:  Apres appel a Echange n="<<n<<" p="<<p<<endl;
}
```

Remarque : L'exécution de ce programme donne:

```
main:  Avant appel a Echange n=10 p=20
Echange: Avant permutation n=10 p=20
Echange: Apres permutation n=20 p=10
main:  Apres appel a Echange n=20 p=10
```

Ici on appelle *Echange* en donnant les adresses de n et p ; celles-ci sont recopiées dans les pointeurs a et b. Le contenu des cases pointées par ces pointeurs est échangé mais a continu de pointé sur n et b sur p. Cette fois la permutation a réussi dans le *main()*. Ceci est donc une méthode si on veut modifier les arguments passés à une fonction. Noter cependant que l'écriture comme l'appel de la fonction *Echange* est très lourd. En C c'est la seule façon de procéder.

6.4 Fonctions et Prototypes

Imaginer que nous ayons à écrire 2 fonctions *F1* et *F2*. Si la fonction *F2* appelle la fonction *F1*, nous devrons écrire

```
int F1(int x)
{
  ...
}
void F2(int x,int y)
{
  int h;
  ...
  h=F1(y);
  ...
}
int main()
{
  int i,j;
  ...
  F2(i,j);
  ...
}
```

En effet comme *F2* appelle *F1*, *F1* doit être déclaré avant *F2* pour que *F1* soit connue dans *F2*. Maintenant supposons que *F2* appelle *F1* et *F1* appelle *F2*...On doit alors utiliser un **prototype**, c'est-à-dire déclarer la fonction *F2*, son type, le nombre de ces arguments et leur type avant:

```
void F2(int,int);  // prototype de F2: ne pas oublier le ";"
int F1(int x)
{
  ...
  if (z<2) F2(x,z);
  ...
}
void F2(int x,int y)
{
  int h;
  ...
  h=F1(y);
  ...
}
int main()
{
  int i,j;
  ...
  F2(i,j);
  ...
}
```

Outre ce cas particulier où il est indispensable, le prototype est utilisé dans tous le fichier ".h" que vous utilisez dans les includes. Nous verrons leur utilité dans l'écriture des classes.

6.5 La surcharge des fonctions

Un des aspects les plus puissants du C++ est que l'on peut "surcharger" les fonctions: c'est à dire que l'on peut donner le même nom à des fonctions qui font des choses différentes. Ce mécanisme s'étend même aux opérateurs (voir le la surdéfinition des opérateurs).

Ce qui permet au langage de distinguer qu'elle est la fonction à appeler, c'est les paramètres demandés lors de l'appel de la fonction.

6.5.1 Exemple

```
#include<iostream>
using namespace std;
//-----------------------------------------
void func(int i)
{
   cout<<"fonction 1 appelée"<<endl;
   cout<<" paramètre = "<<i<<endl;
}
//-----------------------------------------
void func(float i)
{
   cout<<"fonction 2 appelée"<<endl;
   cout<<" paramètre = "<<i<<endl;
} //---------------------------------------
void func(char *s,int i)
{
   cout<<"fonction 3 appelée "<<endl;
   cout<<" paramètre = "<<s<<endl;
   cout<<" paramètre = "<<i<<endl;
}
//-----------------------------------------
int main()
{
   int j=10;
   func(j);
   float k=5.2;
   func(k);
   func("Chaîne",4);
}
```