UN MEILLEUR C

- **Les commentaires**
- **Entrées/sorties avec cin , cout et cerr**
- **Intérêts de cin , cout et cerr**
- **Les manipulateurs**
- **Les conversions explicites**
- **Définition de variables**
- **Variable de boucle**
- **Visibilité des variables**
- **Les constantes**
- **Constantes et pointeurs**
- **Les types composés**
- **Variables références**
- **Allocation mémoire**

LES COMMENTAIRES

Le langage C++ offre un nouvelle façon d'ajouter des commentaires.
En plus des symboles /* et */ utilisés en C, le langage C++ offre les symboles // qui permettent d'ignorer tout jusqu'à la fin de la ligne.
Exemple :

```
/* commentaire traditionnel
   sur plusieurs lignes
   valide en C et C++
*/

void main() { // commentaire de fin de ligne valide en C++
#if 0
     // une partie d'un programme en C ou C++ peut toujours
```

```
     // être ignorée par les directives au préprocesseur
     // #if  ....   #endif
#endif
}
```

Il est préférable d'utiliser les symboles // pour la plupart des commentaires et de n'utiliser les commentaires C (/* */) que pour isoler des blocs importants d'instructions.

ENTREES/SORTIES AVEC *cin, cout* ET *cerr*

Les entrées/sorties en langage C s'effectue par les fonctions *scanf* et *printf* de la librairie standard du langage C.
Il est possible d'utiliser ces fonctions pour effectuer les entrées/sorties de vos programmes, mais cependant les programmeurs C++ préfèrent les entrées/sorties par flux (ou flot ou stream).

Trois flots sont prédéfinis lorsque vous avez inclus le fichier d'en−tête *iostream.h* :

- *cout* qui correspond à la sortie standard
- *cin* qui correspond à l'entrée standard
- *cerr* qui correspond à la sortie standard d'erreur.

L'opérateur (surchargé) **<<** permet d'envoyer des valeurs dans un flot de sortie, tandis que **>>** permet d'extraire des valeurs d'un flot d'entrée.
Exemple :

```
#include <iostream.h>

void main() {
   int i=123;
   float f=1234.567;
   char ch[80]="Bonjour\n", rep;

   cout << "i=" << i << "  f=" << f << "  ch=" << ch;
   cout << "i = ? ";
   cin >> i;          // lecture d'un entier
   cout << "f = ? ";
   cin >> f;          // lecture d'un réel
   cout << "rep = ? ";
   cin >> rep;        // lecture d'un caractère
   cout << "ch = ? ";
   cin >> ch;         // lecture du premier mot d'une chaîne
   cout << "ch= " << ch;  // c'est bien le premier mot ...
```

```
}
/*-- résultat de l'exécution ------------------
i=123  f=1234.57  ch=Bonjour
i = ? 12
f = ? 34.5
rep = ? y
ch = ? c++ is easy
ch= c++
-------------------------------------------------*/
```

- tout comme pour la fonction *scanf*, les espaces sont considérés comme des séparateurs entre les données par le flux *cin*.
- notez l'absence de l'opérateur & dans la syntaxe du *cin*. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

INTERET DE *cin*, *cout* ET *cerr*

- vitesse d'exécution plus rapide : la fonction *printf* doit analyser à l'exécution la chaîne de formatage, tandis qu'avec les flots, la traduction est faite à la compilation.
- vérification de type : pas d'affichage erroné

```
#include <stdio.h>
#include <iostream.h>

void main() {
  int i=1234;
  double d=567.89;
  printf("i= %d   d= %d !!!!!\n", i, d);
  //                  ^erreur: %lf normalement
  cout << "i= " << i << "   d= " << d << "\n";
}
/* Résultat de l'exécution ************
i= 1234   d= -5243 !!!!!!!
i= 1234   d= 567.89
*****************************/
```

- taille mémoire réduite : seul le code nécessaire est mis par le compilateur, alors que pour, par exemple *printf* , tout le code correspondant à toutes les possibilités d'affichage est mis.
- on peut utiliser les flux avec les types utilisateurs (surcharge possible des opérateurs >> et <<).

LES MANIPULATEURS

Les manipulateurs sont des éléments qui modifient la façon dont les éléments sont lus ou écrits dans le flot.
Les principaux manipulateurs sont :

dec	lecture/écriture d'un entier en décimal
oct	lecture/écriture d'un entier en octal
hex	lecture/écriture d'un entier en hexadécimal
endl	insère un saut de ligne et vide les tampons
setw(int n)	affichage de n caractères
setprecision(int n)	affichage de la valeur avec n chiffres avec éventuellement un arrondi de la valeur
setfill(char)	définit le caractère de remplissage
flush	vide les tampons après écriture

Exemple :

```
#include <iostream.h>
#include <iomanip.h>

void main() {
  int i=1234;
  float p=12.3456;

  cout << "|" << setw(8) << setfill('*')
       << hex << i << "|\n" << "|"
       << setw(6) << setprecision(4)
       << p << "|" << endl;
}
/*-- résultat de l'exécution ------------------
|*****4d2|
|*12.35|
-------------------------------------------------*/
```

LES CONVERSIONS EXPLICITES

- En C++, comme en langage C, il est possible de faire des conversions explicites de type, bien que le langage soit plus fortement typé :

```
double d;
int i;

i = (int) d;
```

- Le C++ offre aussi une notation fonctionnelle pour faire une conversion explicite de type :

```
double d;
int i;

i = int(d);
```

- Cette façon de faire ne marche que pour les types simples et les types utilisateurs.
- Pour les types pointeurs ou tableaux le problème peut être résolu en définissant un nouveau type :

```
double d;
int *i;

typedef int *ptr_int;

i = ptr_int(&d);
```

- La conversion explicite de type est surtout utile lorsqu'on travaille avec des pointeurs du type *void* * .

DEFINITION DE VARIABLES

- En C++ vous pouvez déclarer les variables ou fonctions n'importe où dans le code.
- La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.
- Ceci permet de définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité.
  C'est particulièrement utile pour des grosses fonctions ayant beaucoup de variables locales.

Exemple :

```
#include <stdio.hlgt;

void main() {
  int i=0;      // définition d'une variable
  i++;          // instruction
  int j=1;      // définition d'une autre variable
  j++;          // instruction
  int somme(int n1, int n2); // déclaration d'une fonction
  printf("%d+%d=%d\n", i, j, somme(i, j)); // instruction
}
```

VARIABLE DE BOUCLE

On peut déclarer une variable de boucle directement dans l'instruction *for*. Ceci permet de n'utiliser cette variable que dans le bloc de la boucle.
Exemple :

```
#include <iostream.h>

void main() {
  for(int i=0; i<10; i++)
    cout << i << ' ';
  // i n'est pas utilisable à l'extérieur du bloc for
}
/*-- résultat de l'exécution --------
0 1 2 3 4 5 6 7 8 9
-------------------------------------*/
```

VISIBILITE DES VARIABLES

L'opérateur de résolution de portée *::* permet d'accéder aux variables globales plutôt qu'aux variables locales.

```
#include <iostream.h>

int i = 11;

void main() {
    int i = 34;
    {  int i = 23;

       ::i = ::i + 1;
       cout << ::i << "  " << i << endl;
    }
    cout << ::i << "  " << i << endl;
}
/*-- résultat de l'exécution ------------------
12  23
12  34
```

L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (lisibilité). Il est préférable de donner des noms différents plutôt que de réutiliser les mêmes noms.

En fait, on utilise beaucoup cet opérateur pour définir hors d'une classe les fonctions membres.

LES CONSTANTES

Les habitués du C ont l'habitude d'utiliser la directive du préprocesseur *#define* pour définir des constantes.

Il est reconnu que l'utilisation du préprocesseur est une source d'erreurs difficiles à détecter.

En C++, l'utilisation du préprocesseur se limite aux cas les plus sûrs :

- inclusion de fichiers
- compilation conditionnelle.

Le mot réservé *const* permet de définir une constante.

L'objet ainsi spécifié ne pourra pas être modifié durant toute sa durée de vie. Il est indispensable d'initialiser la constante au moment de sa définition.

Exemple :

```
const int N = 10; // N est un entier constant.

const int MOIS=12, AN=1995; // 2 constantes entières

int tab[2 * N];  // autorisé en C++ (interdit en C)
```

CONSTANTES ET POINTEURS

Il faut distinguer ce qui est pointé du pointeur lui même.

- La donnée pointée est constante :

```
const char  *ptr1 = "QWERTY";
ptr1++;      // autorisé
*ptr1 = 'A'; // ERROR: assignment to const type
```

- Le pointeur est constant :

```
char * const ptr2 = "QWERTY";
ptr2++;      // ERROR: increment of const type
*ptr2 = 'A'; // autorisé
```

- Le pointeur et la donnée sont constants :

```
const char * const ptr3 = "QWERTY";
ptr3++;      // ERROR: increment of const type
*ptr3 = 'A'; // ERROR: assignment to const type
```

LES TYPES COMPOSES

En C++, comme en langage C, le programmeur peut définir des nouveaux types en définissant des *struct* , *enum* ou *union*.

Mais contrairement au langage C, l'utilisation de t*ypedef* n'est plus obligatoire pour renommer un type.

Exemple :

```
struct FICHE {        // définition du type FICHE
  char *nom, *prenom;
  int  age;
};

// en C, il faut ajouter la ligne :
//     typedef struct FICHE FICHE;

FICHE adherent, *liste;

enum BOOLEEN { FAUX, VRAI};

// en C, il faut ajouter la ligne :
//     typedef enum BOOLEEN BOOLEEN;

BOOLEEN trouve;

trouve = FAUX;
trouve = 0; // ERREUR en C++ : vérification stricte des types
trouve = (BOOLEEN) 0; // OK
```

VARIABLES REFERENCES

En plus des variables normales et des pointeurs, le C++ offre les variables références.

Une variable référence permet de créer une variable qui est un "synonyme" d'une autre. Dès lors, une modification de l'une affectera le contenu de l'autre.

```
int i;
int & ir = i; // ir est une référence à i
int *ptr;

i=1;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 1  ir= 1

ir=2;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 2  ir= 2

ptr = &ir;
*ptr = 3;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 3  ir= 3
```

Une variable référence doit être initialisée et le type de l'objet initial doit être le même que l'objet référence.

Intérêt :

- passage des paramètres par référence
- utilisation d'une fonction en lvalue

ALLOCATION MEMOIRE

Le C++ met à la disposition du programmeur deux opérateurs *new* et *delete* pour remplacer respectivement les fonctions *malloc* et *free* (bien qu'il soit toujours possible de les utiliser).

- **L'opérateur *new***

  L'opérateur *new* réserve l'espace mémoire qu'on lui demande et l'initialise. Il retourne soit l'adresse de début de la zone mémoire allouée, soit 0 si l'opération à échouée.

```
int *ptr1, *ptr2, *ptr3;

// allocation dynamique d'un entier
ptr1 = new int;

// allocation d'un tableau de 10 entiers
ptr2 = new int [10];
```

```
// allocation d'un entier avec initialisation
ptr3 = new int(10);

struct date {int jour, mois, an; };
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};

// allocation dynamique d'une structure
ptr4 = new date;

// allocation dynamique d'un tableau de structure
ptr5 = new date[10];

// allocation dynamique d'une structure avec initialisation
ptr6 = new date(d);
```

- **L'opérateur *delete***

L'opérateur *delete* libère l'espace mémoire alloué par *new* à un seul objet, tandis que l'opérateur *delete[]* libère l'espace mémoire alloué à un tableau d'objets.

```
// libération d'un entier
delete ptr1;

// libération d'un tableau d'entier
delete[] ptr2;
```

L'application de l'opérateur *delete* à un pointeur nul est légale et n'entraine aucune conséquence facheuse (l'opération est tout simplement ignorée).

A chaque instruction new doit correspondre une instruction *delete*. Il est important de libérer l'espace mémoire dès que celui ci n'est plus nécessaire. La mémoire allouée en cours de programme sera libérée automatiquement à la fin du programme.

- **La fonction *set_new_handler***

Si une allocation mémoire par *new* échoue, une fonction d'erreur utilisateur peut être appelée.

La fonction *set_new_handler* , déclarée dans *new.h*, permet de désactiver la procédure standard (qui est de renvoyer zéro) et d'appeler votre fonction d'erreur.

```
#include <iostream.h>
#include <stdlib.h>   // exit()
#include <new.h>      // set_new_handler()

// fonction d'erreur d'allocation mémoire dynamique
void erreur_memoire( void) {
```

```
  cerr << "\nLa mémoire disponible est insuffisante !!!" << endl;
  exit(1);
}

void main() {
  set_new_handler( erreur_memoire );

  double *tab = new double [1000000000];

  set_new_handler(0); // réactive la fonction d’erreur standard
}
```
