1. **Introduction**

Le C++ est une évolution du langage C conçu par **Bjarne Stroustrup.** Insatisfait des possibilités offertes par le C, il a créé en 1983 le C++ en y ajoutant les possibilités qui, selon lui, manquaient.

Le C++ est un :

* Langage Multi-paradigmes, Il ore en particulier la possibilité de programmer en orienté objet, une technique de programmation puissante.
* Langage de **bas niveau**.
* Langage de programmation très répandu, très utilisé et rapide.

Dans ce chapitre nous insistons sur les différences entre C et C++ et notamment sur tous les éléments nécessaire pour la programmation orientée objet. Cependant des concepts très importants ne sont pas expliquées, pour la raison qu'ils sont réalisées en C++ comme en C, donc supposées acquis.

1. **Premier programme en C++**

Le premier programme C++ à écrire est le suivant :

#include <iostream>

using namespace std ;

int main()

{

int x, y;

cout << "Entrez un nombre entier x: \n" ;

cin >> x ;

cout << "Entrez un autre nombre entier y: \n"

cin >> y ;

cout << "\nla somme de x et y est:"

cout << x+y ;

}

Exemple d’exécution

Entrez un nombre entier x:

10

Entrez un autre nombre entier y:

15

La somme de x et y est 25

**Description**

* ***iostream*:** est la bibliothèque du C++ permettant de faire des entrées sorties. *iostream* contient des déclarations relatives aux flots donc, en particulier, à cin et cout, ainsi qu’aux opérateurs << et >> (dont on verra plus tard qu’ils sont en fait considérés comme des fonctions particulières) ;
* ***cout*:** permet d'afficher des messages à l'écran. Nous verrons ultérieurement que cout est un « flot de sortie » et que << est un opérateur permettant d’envoyer de l’information sur un flot de sortie.
* ***cin :*** permet de lire des données à partir de clavier. nous verrons que cin est un « flot d’entrée » associé au clavier et que << est un opérateur permettant d’« extraire » (de lire) de l’information à partir d’un flot d’entrée.
* ***using namespace std*:** signifie utilisation de l’espace de nommage standard du langage C++. La norme de C++ a introduit la notion d’« espaces de noms » (*namespace*). Elle permet de restreindre la « portée » des symboles à une certaine partie d’un programme et donc, en particulier, de régler les problèmes qui peuvent se poser quand plusieurs bibliothèques utilisent les mêmes noms. Cette notion d’espace de noms sera étudiée par la suite. Pour l’instant, retenez que les symboles déclarés dans le fichier *iostream* appartiennent à l’espace de noms std.
* Un programme écrit en C++ doit être sauvegardé avec l’extension .cpp.

1. **Minimum sur les entrées sorites**

Les entrées et les sorties sont gérées par deux classes **ostream** et **istream** définies dans le fichier d'en-tête **<iostream>**. Un programme qui utilise les flux standard d'entrée-sortie doit comporter la directive : **#include <iostream.h>**

* 1. **Affichage à l’écran**

**Syntaxe** : **cout** < < expr1 < < expr2 < <. . . < < exprn ;

* Cette instruction affiche expr1 puis expr2. . . puis exprn
* **cout** désigne le “flot de sortie” standard.
* **<<** est un opérateur binaire: le premier opérande est cout (de type “flot de sortie”). le second opérande est l’expression à afficher.
* **<<** est surchargé (ou sur-défini): on utilise le même opérateur pour afficher des caractères, des entiers, des réels ou des chaînes de caractères etc.
* Afficher un saut de ligne se fait au moyen de cout << endl.

**Exemples:**

**int** x=10, y = 5;

**cout** << " la somme de "<< x << " et " << y << "est :" <<x+y<< endl;

exp1 exp2 exp3 . . ………….. exp7

Cette instruction affiche: « **la somme de 10 et 5 est : 50** » avec un retour à la ligne (endl)

* 1. **Lecture à partir du clavier**

**Syntaxe** : **cin** > > var1 > > . . . > > varn ;

* Cette instruction lit (au clavier) des valeurs et les affecte à var1 puis var2 . . .puis varn
* **cin** est le flot d’entrée standard, et **>>** est un opérateur similaire à **<<**.
* Les caractères tapés au clavier sont enregistrés dans un buffer dans lequel les **cin** viennent puiser des valeurs. Les espaces, les tabulations et les fins de lignes sont des séparateurs.

**Exemple:**

**int x; float y; char c;**

**cin >> x >> y >> c;**

Cette instruction lit (au clavier) des valeurs et les affecte à x puis y . . .puis z.

1. **Rappel sur les fonctions et la récursivité**

En C++, la partie exécutable d’un programme n’est composée que de fonctions. Chacune de ces fonctions est destinée à effectuer une tâche précise et renvoie généralement une valeur, résultat d’un calcul.

* 1. **Prototypes de fonctions**

En C++, il est obligatoire de déclarer une fonction avant de l'utiliser. Cette déclaration se fait par l'intermédiaire du prototype qui spécifie le nom de la fonction, son type de retour et la liste de ses arguments2.

double agent(int x) ; //

void echange(int \*p, int \*q) ; // quelques prototypes de fonction

void sans() ; //

La signature d'une fonction est composée de son nom et de la liste de ses arguments. Cette notion importante sert à différencier entre eux les appels d'une fonction surchargée (c.f. 1).

agent(int x) // signature de la fonction agent

echange (int \*p, int \*q) // signature de la fonction echange

sans() // signature de la fonction sans

* 1. **Définition de fonctions**

Une fonction est caractérisée par : Son nom, le type de valeur qu’elle renvoie (type de retoure), ses paramètres (l’information qu’elle reçoit pour faire son travail), l’instructions-bloc qui effectue le travail (corps de la fonction).

La définition d’une fonction est de la forme suivante :

**type** **nom** (**liste de paramètres formels**)

**{**

<instructions-bloc>

**}**

Donnons par exemple les définitions des trois fonctions déclarées ci-dessus :

double Moyenne(double x, double y)

{ return (x + y) / 2.0;

}

char LireCaractere()

{ char c;

cin >> c;

return c;

}

void AfficherValeurs(double valeur)

{

cout << ’\t’ << valeur << ’\n’;

}

L’utilisation d’une fonction se fait grâce à l’appel de la fonction. Cet appel est une expression de la forme : **nom** (liste d’expressions). Voici un bout de programme avec appel des trois fonctions précédentes :

**int** main ()

**{**

**double** x, y;

cout << "\n Entrez les valeurs de x et y :";

cin >> x >> y;

**double** m = Moyenne(x, y);

cout << "\n Voulez-vous afficher la moyenne ? ";

if (LireCaractere() == ’o’)

AfficherValeurs(m);

**}**

* 1. **Récursivité**
     1. **Notion de récursivité**

La récursivité est une manière naturelle et souvent élégante pour écrire de nombreux algorithmes. Elle consiste à décrire les étapes nécessaires à la résolution d’un problème en utilisant la résolution du même problème, sur des entrées plus petites.

Un algorithme (fonction ou procédure) récursif est un algorithme qui fait appel à lui-même dans le corps de sa propre définition.

Dans un algorithme récursif, on nomme appel récursif toute étape de l’algorithme résolvant le même problème sur une autre donnée.

* + 1. **La forme générale d’une fonction récursive:**

type nom (……….)

{

if ( condition de fin)

{

// pas d’appel récursif

}

else

{….

f(…) // la fonction f appelle elle même une ou plusieurs fois

………..

}

}

* + 1. **Exemple**

La factorielle d’un nombre n donné est le produit des nombres entiers inférieurs ou égales à ce nombre.

Il existe différente façon de le décrire :

0 !=1

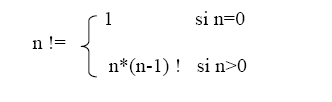
1 !=1

2 !=1\*2

.

.

n !=1\*2\*………….\*(n-1)\*n

Une deuxième notation plus rigoureuse fait appel à la récurrence.

La fonction factorielle(n) permet de calculer la factorielle de l’entier n. cette fonction est récursive et se rappelle une fois en factorielle (n-1). Il y a fin d’appels récursifs lorsque n vaut 0.

**#include <iostream>**

**using namespace std;**

**int factorielle(int n) {**

**if (n == 0)**

**return 1;**

**else**

**return n \* factorielle(n - 1);**

**}**

**int main() {**

**cout << " 4! = " << factorille(4) << endl;**

**}**

Si on considère que n = 4. La factorielle de 4 est calculée de la manière suivante :

**factorial(4) = 4 \* factorielle(3)**

**= 4 \* 3 \* factorielle(2)**

**= 4 \* 3 \* 2 \* factorielle(1)**

**= 4 \* 3 \* 2 \* 1 \* factorielle(0)**

**= 4 \* 3 \* 2 \* 1 \* 1**

**= 4 \* 3 \* 2 \* 1**

**= 4 \* 3 \* 2**

**= 4 \* 6**

**= 24**

La figure suivant la séquence d’appel pour f**actorielle(4)** invoqué par la fonction **main**.

main ( )

2

2

1

0

**factorielle(2)**

**factorielle(4)**

**factorielle(0)**

24

**factorielle(3)**

6

4

3

Retour de résultat

Appel récursif

2

1

**factorielle(1)**

* + 1. **Règles de conception**

**Règle 1 :** Tout algorithme récursif doit distinguer plusieurs cas, dont l’un au moins ne doit pas comporter d’appel récursif, sinon il y aura un risque de cercles vicieux et de calcul infini. Les cas non récursifs d’un algorithme récursif sont appelés cas de base ou cas triviaux. Les conditions que doivent satisfaire les données dans ces cas de base sont appelées conditions de terminaison.

**Règle 2 :** Tout appel récursif doit se faire avec des données plus « proches » de données satisfaisant une condition de terminaison, et il doit donc en principe se « rapprocher » d’un cas de base, de façon à permettre la terminaison du programme.

1. **Pointeurs**

Quand on déclare une variable avec un nom et un type, un emplacement mémoire du type de la variable est créé à une certaine adresse avec son nom pour y accéder. L’emplacement mémoire recevra la valeur de la variable lors d’une affectation.

int x ; // une déclaration

x = 0 ; // une affectation

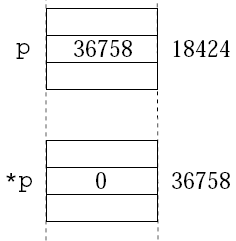
Le C permet de manipuler dans le programme, les adresses des emplacements mémoire des variables. &x désigne l’adresse de la variable x. Un ***pointeur*** est une variable qui contient l’adresse d’une autre variable. La déclaration d’un pointeur est de la forme suivante : <type> \* <nom> ;

Exemple :

int \*p; // pointeur sur un entier

char \*s ; // // pointeur sur un entier

* 1. **L'opérateur &:**

L'opérateur & sert à récupérer l'adresse mémoire d'une variable. Dans l'exemple ci-dessous, on met l'adresse mémoire de variable x dans le pointeur p:

Exemple :

int x=0;

**x**

int \*p;

P = & x ; // p prend l’adresse de x (36758).

La variable p est alors un pointeur sur un int ; la variable entière dont p contient l’adresse est dite pointée par p.

* 1. **L'opérateur \*:**

Un pointeur contient une adresse en mémoire, et grâce à l'opérateur \*, on peut accéder à la valeur contenue à cette adresse (soit pour la modifier, soit pour l'examiner).

Exemple :

int x = 5; int \*p;

p = &x;

cout << "La valeur de x est" <<\*p;  // affiche 5

L'exemple ci-dessous illustre comment on peut modifier le contenu d'une variable indirectement, par son pointeur :

float y = 5; float \*p;

p = &y;

\*p = 4;

cout << "La valeur de x est" <<\*p; // affiche 4

* 1. **Allocation / Désallocation dynamiques de l’espace mémoire en C++**

Contrairement aux variables déclarées statiquement, c’est-à-dire celles déclarées au début du programme explicitement par leurs noms et leurs types, les variables crées dynamiquement peuvent être créées **au fur et à mesure des besoins.** En revanche, elles permettent aussi de récupérer de l’espace mémoire après leur utilisation, ce qui permet une gestion très rationnelle de la mémoire centrale.

L’allocation dynamique en C++ se fait en utilisant le mot clé **‘new’** suivi du type de la variable à créer. L’espace alloué est généralement affecté à un pointeur permettant sa manipulation. Une fois la variable créée n’est plus utilisée, il est possible de désallouer l’espace mémoire occupé par celle-ci en utilisant le mot clé **‘delete’** suivi de son adresse (indiquée par un pointeur). Voici quelques exemples d’utilisation:

Exemple 1 : allocation d’un seul objet

int \* pt=**new** (int) ; // déclaration avec allocation de l’espace mémoire pour un entier

\*pt= 15 ;

cout <<  (\*pt) ; // affiche 15

. . .

**delete** (pt) ; // libération de l’espace mémoire

Exemple 2 : allocation d’un tableau d’objet

float \*T=**new** float [100] ; // allocation de l’espace mémoire pour un tableau de 100 réel.

……….

**delete** [ ] T; // libération de l’espace mémoire réservé pour un tableau de 100 réel

**A retenir :**

*Grace aux pointeurs, on peut manipuler des structures de données dynamiques dont la taille n’est déterminée qu’au moment de l’exécution.*

1. **Références**

A coté des pointeurs, les références sont une autre manière de manipuler les adresses des objets placés dans la mémoire. Une référence est une variable qui coïncide avec une autre variable. La déclaration d’une référence est de la forme suivante : ***<type> & <nom> = <nom var>;***

**Exemple :**

int n = 10;

int &r = n; // r est une référence sur n

Cela signifie que r et n représentent la même variable (en particulier, elles ont la même adresse). Si on écrit par la suite : r = 20, n prend également la valeur 20.

**Attention :**

double somme, moyenne;

double &total = somme; // correct : total est une référence sur somme

double & r; // illégal : pas de variable à laquelle se référer

double &total = moyenne; // illégal : on ne peut redéfinir une référence

Une valeur de type référence est une adresse mais, hormis lors de son initialisation, toute opération effectuée sur la référence agit sur l'objet référencé, non sur l'adresse. Il en découle qu'il est obligatoire d'initialiser une référence lors de sa création ; après, c'est trop tard.

1. **Spécificités de C++ sur le langage de C**

Le C++ dispose d’un certain nombre de spécificités par rapport a C qui ne sont pas axées sur l’orienté objet :

* Le commentaire en fin ligne
* L’emplacement libre des déclarations
* La sur-définition de fonction
* Les opérateurs *new* et *delete*
* Les arguments par défaut
* Les fonctions en ligne
* La programmation orientée objet
  1. **Commentaires en fin ligne**

Pour écrire des commentaires dans un programme, le programmeur dispose toujours des commentaires C avec /\* et \*/ et aussi de commentaires en fin de ligne avec //.

cout << " bonjour " ; // ceci est une formule de politesse

* 1. **Placement des déclarations des variables**

En C les déclarations de variables doivent apparaitre au début d'un bloc. En C++, au contraire, on peut mettre une déclaration de variable partout où on peut mettre une instruction. Cette différence parait mineure, mais elle est importante par son esprit. Elle permet de repousser la déclaration d'une variable jusqu'à l'endroit du programme où l'on dispose d'assez éléments pour l'initialiser. On lutte aussi contre les variables déjà déclarées mais non encore initialisées qui sont un important vivier de bugs dans les programmes.

**Exemple :** un programme qui calcule la somme d’un tableau d’entier

Version C :

int main()

{

int i, int s=0 ; int T [10] ; // déclaration de tous les variables utilisés

for (i=0 ; i<10 ; i=i+1)

scanf("%d", &T[i]) ;

for (i=0 ; i<10 ; i=i+1)

s= s+ T[i];

printf("la somme est :%d", s) ;

}

Version C++ :

{

Int T [10] ;

for (int i=0 ; i<10 ; i=i+1) // déclaration de la variable i dans la boucle for

scanf("%d", &T[i]) ;

int s=0 ; // déclaration et initialisation de s directement //avant son utilisation.

for (i=0 ; i<10 ; i=i+1)

s= s+ T[i];

printf("la somme est :%d", s) ;

}

* 1. **Type bool**

En plus des types définis par l'utilisateur (ou classes, une des notions fondamentales de la programmation orientée objets) C++ possède quelques types qui manquaient à C, notamment le type booléen et les types références. Le type ***bool*** (pour booléen) comporte deux valeurs : ***false*** et ***true***. Contrairement à C où on utilise des entiers.

Exemple :

bool b ;

….

if (b==true) // on peut écrire aussi if (b)

…

if (b==false) // on peut écrire aussi ⬄ if (!b)

* 1. **Passage de paramètre par références**

En C, les arguments sont passés par valeur. Ce qui signifie que les paramètres d’une fonction C sont toujours en entrée de la fonction et pas en sortie de la fonction ; autrement dit les paramètres d’une fonction ne peuvent pas être modifiés par la fonction. Ci-dessous la fonction echange est censée échanger les valeurs des deux paramètres.

void echange(int a, int b) {

int c = a ;

a = b ;

b = c ;

}

int main() {

int x = 10 ; int y = 20 ;

cout << " avant appel : " << x << " " << y << endl ;

echange(x, y) ;

cout << " apres appel : " << x << " " << y << endl ;

}

Malheureusement, pour la raison invoquée plus haut, la sortie de ce programme est :

avant appel : 10 20

apres appel : 10 20

Les programmeurs C ont l’habitude de palier à cet inconvénient du C en passant l’adresse des paramètres en sortie d’une fonction au lieu de passer la valeur. Ainsi, notre exemple sera écrit:

void echange(int \* a, int \* b) {

int c = \*a ;

\*a = \*b ;

\*b = c ;

}

int main() {

int x = 10 ; int y = 20 ;

cout << " avant appel : " << x << " " << y << endl ;

echange(x, y) ;

cout << " après appel : " << x << " " << y << endl ;

}

La sortie de ce programme correspond à ce que notre intuition attend :

avant appel : 10 20

apres appel : 20 10

Cette manipulation de pointeurs est lourde à gérer pour le programmeur. C++ palie à cet inconvénient en permettant le *passage de paramètres par référence* qui est équivalent au passage par variable en algorithmique (var ⬄ &); la programmation devient plus légère et le résultat correct. Il suffit de changer la déclaration de la fonction echange :

void echange(int & a, int & b)

{ int c = a ;

a = b ;

b = c ;

}

int main() {

int x = 10 ; int y = 20 ;

cout << " avant appel : " << x << " " << y << endl ;

echange(x, y) ;

cout << " après appel : " << x << " " << y << endl ;

}

La sortie de ce programme correspond à ce que notre intuition attend :

avant appel : 10 20

apres appel : 20 10

L’appel de la fonction est le même que pour le passage par valeur : echange(x, y) ;

On notera qu’il n’y a rien d’extraordinaire. Pascal possède cette propriété de passage de paramètres par référence et C++ ne fait que combler un manque important du C.

* 1. **Surcharge des noms des fonctions**

A condition qu'elles se distinguent par leur signature, en C++ plusieurs fonctions peuvent avoir le même nom. A l'appel de la fonction, le choix de la fonction surchargée se fera en fonction du nombre et du type des arguments servant à l'appel.

Exemple:

**int** min(int a, int b){

if (a>b) return a;

else return b;

}

**int** min(int a, int b, int c){

**return** min(a, min(b, c)) ; // attention, ici ce n'est pas un appel récursif

} // on utilise juste la fonction précédente

**char** **\*** min(char \*s1, char \*s2)

{

**return** strcmp(s1,s2) < 0 ? s1 : s2 ;

}

int main ()

{

cout << "le min de 25 et 37 est :"<< min (25, 37) << endl;

cout << "le min de 10.5 et 16.6 est :"<< min (10.5, 16.6) << endl;

cout << "le min de F et et S est :"<< min ('F', 'S') << endl;

}

* 1. **Les opérateurs new et delete**

Des différences entre C et C++ existent aussi au niveau de l'allocation et de la restitution dynamique de mémoire.

Les fonctions *malloc* et *free* de la bibliothèque standard C sont disponibles en C++. Mais il est fortement conseillée de leur préférer les opérateurs *new* et *delete*. La raison principale est la suivante :

* les objets crées à l'aide de new sont initialisés a l'aide des constructeurs correspondants, ce que ne fait pas *malloc*.
* De même, les objets libérés en utilisant *delete* sont finalisés en utilisant le destructeur de la classe correspondante, contrairement à ce que fait free.