

Chapitre 1

INTRODUCTION AU LANGAGE C++

Plan du chapitre :

I. Historique et présentation	2
II. Implémentation des modules en C++	3
III. Nouvelles possibilités en C++	4

Objectifs du chapitre :

- Présenter le langage C++.
- Connaître les nouvelles possibilités apportées par le langage C++ par rapport au langage C indépendamment de la programmation orientée objet.

I. Historique et présentation:

I.1 Historique :

En 1972, **Dennis Ritchie** a conçu un langage de programmation structurée pour développer une version portable du système d'exploitation UNIX dans les « *Bell Labs* » : **Langage C**.

En 1982, **Bjarne Stroustrup** a intégré la programmation orientée objet au langage C dans les laboratoires d'*AT&T Bell*, d'où le successeur de C : le **langage C++** qui est en fait une surcouche du C (C++ signifie une incrémentation du C).

C++ n'était pas le premier langage de programmation qui a intégré la programmation orientée objet, en effet il y avait :

- **Simula (Simple universal language)** est le langage qui a introduit le paradigme orienté objet en programmation, en 1960, et il est donc considéré comme le premier langage à objet et le prédécesseur de langages ultérieurs tels que Smalltalk et C++.
- **Smalltalk** est un des premiers langages de programmation objet. Il a été créé en 1972.

I.2 Présentation :

Le langage C++ est un langage de programmation évolué et structuré, il est considéré comme un successeur de C. Tout en gardant les points forts de ce langage, il corrige certains points faibles et permet l'abstraction de données. C'est un langage à **typage fort**, **Compilé** (impératif) et **orienté objet** (POO) (il reprend aux trois principes fondamentaux : encapsulation, polymorphisme et héritage).

Remarquons toutefois que C++ n'est pas purement objet (comme le sont par exemple *Eiffel* ou *Smalltalk*) mais est un langage hybride: on peut très bien programmer en C++ sans pour autant programmer par objets

Avantages :

Les principaux avantages du C++ sont les suivants :

- Grand nombre de fonctionnalités ;
- Performances du C ;
- Portabilité des fichiers sources ;
- Facilité de conversion des programmes C en C++, et en particulier, possibilité d'utiliser toutes les fonctionnalités du langage C ;
- Applications efficaces grâce à la compilation.
- Compilateur disponible sur pratiquement toutes les plates-formes et documentation nombreuse, grâce à sa large diffusion

Inconvénients :

- Effets indésirables et comportement peu intuitif, conséquence de la production automatique de code.
- Syntaxe parfois lourde et peu naturelle.
- Le langage ne définit pas de techniques de récupération automatique de mémoire (*garbage collector*) comme c'est le cas en JAVA.

II. Implantation des modules en C++ :

Tout comme en C, un module C++ est généralement implanté par deux fichiers : un fichier d'interface qui regroupe les services offerts par le module (définition de types, constantes, déclaration de fonctions) et un fichier d'implantation qui permet de définir les fonctions déclarées dans l'interface. Les extensions conventionnellement utilisées en C++ sont :

- **.hh, .H** pour les fichiers d'interface C++ (les *headers*). Il n'est pas rare cependant de voir des fichiers d'extension **.h** contenir des interfaces C++.
- **.cc, .cpp, .C** pour les fichiers d'implantation C++.

Structure d'un programme C++ :

La structure minimale d'un programme C++ est similaire à celle d'un programme C. Elle peut ensuite être étendue par des éléments (fonctions, instructions, structures de contrôle, etc...) abordés lors de l'étude du langage C, et par les éléments propres au C++ présentés dans ce polycopié.

Pour pouvoir être compilé en une séquence binaire exécutable, le code source doit fournir au compilateur un «point d'entrée». Par convention, ce point d'entrée est en C++ une fonction intitulée **main** qui peut être définie de deux manières différentes :

// Programme sans paramètre :

```
int main() { ... } //
```

// Programme avec paramètres :

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

argc : nombre de paramètres à passer dans main.

argv : tableau de paramètres, argc entrées dont le premier contient la chaîne donnant le nom du programme.(exemple: "C:\Programme Files\...\Hello.exe ")

Note : La fonction main doit renvoyer un code d'erreur d'exécution du programme, le type de ce code est int. Elle peut aussi recevoir des paramètres du système d'exploitation. La valeur 0 retournée par la fonction main indique que tout s'est déroulé correctement.

Exemple:

```

Hello.CPP
#include <iostream.h>
void main()
{
    cout <<"Hello World!"<<
}

```

```

Hello.exe
01001001010001011
00001010101000000
00010010011101010
10010000000000001
...

```

III. Nouvelles possibilités du C++ :

Le langage C++ est un sur-ensemble du langage C. Conséquence : un compilateur C++ est capable de compiler un texte C. Il ne s'agit malheureusement pas d'un sur-ensemble parfait et il existe entre les deux langages quelques incompatibilités mineures.

Indépendamment des apports permettant la programmation par objets, le langage C++ propose des améliorations au langage C. A dire vrai, plusieurs de ces améliorations se justifient par les besoins particuliers de la programmation par objets, mais peuvent être étendues sans complication à la programmation traditionnelle.

III.1. Les commentaires :

Il existe deux types de commentaires en C++ : les commentaires de type C et les commentaires de fin de ligne qui ne sont disponibles qu'en C++.

Exemple:

```
void main()
{
    /* ceci est un commentaire en C */
    // ceci est un commentaire en C++
}
```

III.2. Types de base :

Les types de bases disponibles en C, tels que `char`, `int`, `float`, `double`, `void` sont également disponibles en C++. Un type supplémentaire a été introduit pour manipuler de manière plus rigoureuse et plus explicite les booléens, c'est le type `bool`. Les variables de type `bool` peuvent avoir deux valeurs différentes : `true` ou `false`.

```
bool stop = false;
while (!stop) {
    ...}
```

III.3. Les constantes:

C++ offre la possibilité de définir des entités constantes (variables, fonctions). La valeur de ces entités ne peut alors plus être modifiée lors de l'exécution du programme, ce qui suppose qu'elles doivent être initialisées à la déclaration.

```
int main()
{
    const int a; //Erreur à la compilation !
                //error C2734:'a': const object must be initialized
    const int b=5; // correct
    b=2 ;        // error C2166: l-value specifies const object
}
```

Remarque:

On fera une distinction bien nette entre les constantes de compilation définies avec la directive `#define` du préprocesseur et les constantes définies avec le mot clé `const`. En effet, les constantes littérales ne réservent pas de mémoire. Ce sont des valeurs immédiates, définies par le compilateur et elle conservera toujours sa valeur. En revanche, les variables de classe de stockage `const` peuvent malgré tout avoir une place mémoire réservée. Ce sont donc plus des variables accessibles en lecture seule que des constantes. Par ailleurs, les constantes littérales n'ont pas de type, ce qui peut être très gênant et source d'erreur.

III.4. Déclaration des variables :

Contrairement au C qui impose de définir toutes les variables d'un bloc **au début de ce bloc**, C++ offre la possibilité de définir les différentes variables utilisées au fur et à mesure des besoins. Si une variable locale est déclarée au début d'un bloc, sa portée est limitée à ce bloc.

Exemple:

```
void main()
{
    for (int i=0;i<10;i++ )//i est connue dans toute la
    {                               // fonction «main()»
        int j; //j n'est connue que dans le bloc du for
        ...;
    }
}
```

III.5. Les entrées-sorties :

On peut utiliser en C++ les fonctions d'entrée/sortie classiques du C (printf, scanf, puts, gets, putc, getc ...), à condition de déclarer le fichier d'en-tête stdio.h.

Il existe de nouvelles possibilités en C++ qui sont plus simples à utiliser, à condition de déclarer le fichier d'en-tête **iostream.h**. Ces nouvelles possibilités ne nécessitent pas de FORMATAGE des données.

➤ La sortie standard "**cout**" :

cout permet d'afficher, sans formatage :

- Des entiers
- Des réels
- Des caractères
- Des chaînes de caractères (variables ou constantes)
- ...

Utilisation : cout<<objet à afficher

Exemple:

```
//include indispensable pour cout
#include <iostream.h>
void main()
{
    cout << "Hello World !";
    cout << "Hello World !\n";
    cout << "Hello World !" << endl;
    int n = 5;
    cout << "La valeur est " << n << endl;
    float f = 3.14f;
    char *ch = "Coucou";
    cout << ch << " float = " << f << endl;
}
```

Ce programme donne en sortie :

```
Hello World !Hello World !
Hello World !
La valeur est 5
Coucou float = 3.14
```

Le "**endl**" est en fait disponible pour éviter d'éventuels "\n", en fin de ligne.

➤ *L'entrée standard "cin" :*

cin permet de saisir, sans formatage, et sans utiliser l'opérateur d'adressage & :

- Des entiers
- Des réels
- Des caractères
- Des chaînes de caractères
- ...

Utilisation : cin >> objet à saisir

Exemple:

```
//include indispensable pour cout et cin
#include <iostream.h>
void main()
{
    int n;
    cout << "Entrez un entier : ";
    cin >> n;
    cout << "Vous avez entré : " << n << endl;
    char ch[81]; float f;
    cout << "Entrez un entier, une chaîne, puis un float:";
    cin >> n >> ch >> f;
    cout << "Vous avez entré : " << n << ch << f << endl;
}
```

Cet exemple illustre brièvement comment fonctionne "cin". Bien entendu, aucun contrôle de type n'est effectué, c'est donc à l'utilisateur qu'il advient de faire attention.

Attention:

```
int *p, n; p = &n;
cin >> *p; // pour la saisie d'un pointeur
int *tab, i;
tab = (int*) malloc(100 * sizeof(int)); // tableau
for(i=0; i<100; i++)
    cin >> *(p+i); // pour la saisie d'un tableau
```

III.6 Les arguments par défaut pour les fonctions:

En C++, on peut préciser la valeur prise par défaut pour un argument de fonction. Lors de l'appel de cette fonction, si on ne met pas un argument, il prendra la valeur indiquée par défaut, sinon cette valeur par défaut est ignorée.

Exemple:

```
void f1(int a, int b=3) // par défaut le paramètre b vaut 3
{
    ....;
}
void main()
{
    int x=2, y=5;
    f1(x);
    // a prendra 2 et b prend la valeur par défaut 3;
    f1(x, y);
    // a=2, b=5: l'initialisation par défaut est ignorée
    f2(); // interdit
}
```

Remarques :

a. Une fonction peut définir des valeurs par défaut pour tous ses paramètres ou seulement pour une partie. Les paramètres acceptant des valeurs par défaut doivent se trouver *après* les paramètres sans valeur par défaut dans la liste des paramètres acceptés par une fonction.

La déclaration suivante est interdite:

```
void f4(char c='a', int n)
```

b. Les valeurs par défaut de chaque paramètre ne peuvent être mentionnées qu'une seule fois parmi les définitions / déclarations d'une fonction. Ainsi, par convention, ces valeurs sont généralement mentionnées dans la *déclaration* de la fonction et pas dans sa *définition* (donc dans le .H et pas dans le .C).

III.7. Surcharge (surdéfinition) des fonctions :

En C++, il est possible, que dans un même programme, plusieurs fonctions possèdent le même nom.

La surcharge permet d'attribuer le même nom à plusieurs fonctions qui se différencient par le type et/ou le nombre de leurs arguments.

Cela permet par exemple de déclarer dans un même programme les fonctions présentées:

Exemple:

```
#include <conio.h>
#include <math.h>
void affiche(int x)
{
    cout<<x;
}
void affiche(float z)
{
    cout<<z;
}
void affiche(char *chaine)
{
    cout<<chaine ;
}
void main()
{
    int d = 10;
    float pi = atan(1) * 4;
    affiche(d);
    affiche(pi);
    affiche("Fin de l'exemple");
}
```

Remarques :

- Ceci facilite donc la tâche du programmeur, qui peut associer un nom unique à une action déterminée, quels que soient les arguments fournis. La fonction correspondant aux paramètres fournis est alors exécutée.
- Attention aux ambiguïtés :

Exemple:

```
void f(float a, double b)
{
    ...
}
void f(double a, float b)
{
    ...
}
void main()
{
    float x, y ;
    f(x, y) ;//erreur d'ambigüité d'appelle
```

III.8. Les opérateurs NEW et DELETE:

En plus des "anciens" **malloc** et **free** du C, C++ possède un nouveau jeu d'opérateurs d'allocation/désallocation de mémoire : **new** et **delete**.

Ils ont été créés principalement pour la gestion dynamique des objets, mais on peut les utiliser également pour des variables simples. Voici une comparaison d'utilisation :

<pre>... /* pour un simple pointeur */ int * pInt; pInt = (int*)malloc(1*sizeof(int)); free(pInt); ... /* pour un tableau */ pInt = (int*)malloc(100*sizeof(int)); free(pInt); ...</pre>	<pre>... // pour un simple pointeur int * pInt; pInt = new int; delete pInt; ... // pour un tableau pInt = new int[100]; delete[] pInt; ...</pre>
--	---

Remarque :

- Lorsqu'il n'y a pas assez de mémoire, new renvoie un pointeur nul.
- Il faut utiliser delete[] avec new[] et delete avec new.
- Il ne faut pas mélanger les mécanismes d'allocation mémoire du C et C++ (utilisation de delete sur un pointeur renvoyé par malloc).

III.9. Les références:

III.9.1. Définition:

C++ introduit une nouvelle notion fondamentale : les références. Les *références* sont des synonymes d'identificateurs.

Par exemple, si « *id* » est le nom d'une variable, il est possible de créer une référence « *ref* » de cette variable.

Les deux identificateurs *id* et *ref* représentent alors la même variable, et celle-ci peut être accédée et modifiée à l'aide de ces deux identificateurs indistinctement.

La syntaxe de la déclaration d'une référence est la suivante :

```
type &référence = identificateur;
```


Exemple:

```
int i=0;
int &ri=i;    // Référence sur la variable i.
ri=ri+i;     // Double la valeur de i (et de ri).
cin >>ri ;
```

Remarque :

- On peut créer plusieurs références sur la même variable :

```
float z ;
float &r1=z, &r2=z;
```

III.9.2. Pointeurs et références :

Les références et les pointeurs sont étroitement liés. En effet, une variable et ses différentes références ont la même adresse, puisqu'elles permettent d'accéder à un même objet. Utiliser une référence pour manipuler un objet revient donc exactement au même que de manipuler un pointeur constant contenant l'adresse de cet objet. Les références permettent simplement d'obtenir le même résultat que les pointeurs, mais avec une plus grande facilité d'écriture.

Exemple:

<pre>// Pointeur int i=0; int *pi=&i; *pi=*pi+1; // Manipulation // de i via pi</pre>	<pre>//Référence int i=0; int &ri=i; ri=ri+1; // Manipulation // de i via ri</pre>
--	---

Comparaison:

- Les références doivent être initialisées.
- On ne peut pas faire un déréférencement pour les références.

<pre>int a=2,b; int &ra=a; int &ra=b; faux</pre>	<pre>int *p,a=2,b; p=&a; p=&b ;</pre>
---	---

- On peut affecter à une référence un emplacement dynamique :

```
int &x= *new int ;
int &y= *new int(5);
```

5

- Pas de parcourt dynamique avec les références.

<pre>//Pointeur int *p=new int[10] ; //parcourt de 10 cases avec p for(i=0 ;i<10 ;i++) { cin >> *(p+i) ; ...</pre>	<pre>//Référence int i=0; int &ri=*new int[10]; //parcourt de 10 cases avec ri for(i=0 ;i<10 ;i++)</pre>
---	--

☞ Une référence reste toujours liée à l'emplacement mémoire initial.

- Les références sont plus faciles à manipuler que les pointeurs.

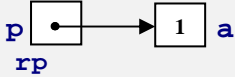
III.9.3. Référence à un pointeur :

Lorsqu'on doit transmettre en argument la référence à un pointeur, on est amené à utiliser ce genre d'écriture :

```
int *&adr ; //adr est une référence à un pointeur sur int
```

Exemple:

```
int a=1;
int *p=&a;
int *&rp=p; // référence à p
```



Remarque :

- Pour changer l'adresse d'un pointeur à travers une fonction il faut le passer par référence

```
int x;
void f(int *&adr )
{ adr=&x ; }
void main(){
int *p=&a ;
f(p) ; //après l'appel de f p ne pointera
// jamais sur a mais c'est sur x
```

III.9.4. Passage de paramètre par référence en C++ :

La notion de référence est directement liée au passage de paramètres à des fonctions en C. Nous savons que lorsque nous voulons transmettre à une fonction la valeur d'une variable ou au contraire la donnée réelle (en fait l'adresse), nous n'utilisons pas les mêmes méthodes. Par exemple, soit les fonctions suivantes :

- *Mode de passage par valeur :*

```
void permut_v(int a, int b)
{
    int x=a ;
    a=b ;
    b=x ;
}

void main()
{
    int i=5,j=3 ;
    permut_v(i,j) ;
    cout << « i= » <<i<<endl<< « j= » <<j ;
}
```

main		permut_v
i=5, j=3	Appel →	a=5, b=3
		↓ Exécution
i=5, j=3	Retour ←	a=3, b=5
les valeurs des variables i et j de la fct main ont été copiées dans les paramètres a et b de la fonction permut_v, la modification de a et b n'altère en aucun cas les valeurs de i et j		en mémoire a et b sont distinctes de i et j, les paramètres formels ont simplement reçu une copie des valeurs des variables i et j

- *Mode de passage par adresse :*

```
void permut_a(int *a, int *b)
{
    int x=*a ;
    *a= *b ;
    *b=x ;
}

void main()
{
    int i=5,j=3 ;
    permut_a(&i,&j) ;
    cout<< « i= »<<i<<endl<< « j= »<<j ;
}
```

main		permut_a
i=5, j=3	Appel →	a=&i, b=&j, *a=5, *b=3
		↓ Exécution
i=3, j=5	Retour ←	*a=3, *b=5 → i=3, j=5
la permutation a bien eu lieu car la fonction main a transmis à la fonction permut_a les adresses des variables i et j aux pointeurs a et b.		a contient l'adresse de i et b contient l'adresse de j → la modification de *a et *b modifie systématiquement les variables i et j de la fonction main

- *Mode de passage par référence :*

```
void permut_r(int &a, int &b)
{
    int x=a ;
    a=b ;
    b=x ;
}

void main()
{
    int i=5,j=3 ;
    permut_r(i,j) ;
    cout<< « i= »<<i<<endl<< « j= »<<j ;
}
```

main		permut_r
i=5, j=3	Appel →	a est une référence à i, et b est une référence à j → a=5, b=3
		↓ Exécution
i=3, j=5	Retour ←	a=3, b=5
la permutation a bien eu lieu car en appelant la fonction permut_r, c'est comme si on avait la déclaration : int &a=i, int &b=j		a et i occupent la même case mémoire, ainsi que b et j → la modification de a et b implique automatiquement la modification de i et j

Le "&" signifie que l'on passe par référence. La ligne de déclaration de la fonction est en fait la seule différence avec une transmission par valeur, d'un point de vue code. C'est-à-dire que l'utilisation des variables dans la fonction s'opère sans "*" et l'appel à la fonction sans "&". C'est ce qui fait la puissance des références : c'est transparent pour l'implémentation, mais cela possède la puissance des pointeurs.

Note : Il est recommandé, pour des raisons de performances, de passer par référence tous les paramètres dont la copie peut prendre beaucoup de temps (en pratique, seuls les types de base du langage pourront être passés par valeur).

III.9.5. Référence sur des constantes:

Il est possible de faire des références sur des constantes :

```
const int &ri=3; // Référence sur 3.      ri 3
```

Remarque:

- La transmission par référence impose de passer un paramètre de même type que l'argument.

Exemple:

```
void f(int &n); // f reçoit la référence à un entier
float x ;
...
f(x) ; // appel illégal
```

- ✚ Pour accepter un paramètre par référence de type différent il suffit d'ajouter *const* à l'argument.

```
void f(const int &n); // f reçoit la référence à un entier
float x ;
f(x) ; // correct : f reçoit la référence à une variable
//temporaire contenant le résultat de la conversion
//de x en int
```

- Supposons qu'une fonction *fct* ait pour prototype:

```
void fct(int &n);
```

Le compilateur refusera alors un appel de la forme suivante :

```
fct(3);
//de même pour
const int c=3 ;
fct(c); // incorrect fct ne peut pas modifier une constante
```

- ✚ La déclaration *const int &* correspond à une référence à une constante.

```
void fct(const int &n); // dans fct on n'a pas le droit de
changer n
...
const int c=15 ;
fct(3) ; // correct
fct(c) ; // correct
```

III.9.6. Transmission par référence d'une valeur de retour:

Le mécanisme que nous venons d'exposer pour la transmission des arguments s'applique à la valeur de retour d'une fonction. Considérons ce petit exemple :

```
int &f()
{ ...
  return n ;
}
```



Un appel de *f* provoquera la transmission en retour non plus d'une valeur, mais de la référence de *n*.

```
int p
...
p =f(); //affecte à p la valeur située à la
        //référence fournie par f
```

Il est nécessaire que *n* ne soit pas locale à la fonction, sous peine de récupérer une référence à quelque chose qui n'existe plus ! D'out il devient possible d'utiliser son appel comme une variable :

```
int x;
f()=2*x;
```

III.10. Les variables statiques:

Le mot clé ***static*** signifie que la variable a une durée statique (elle est allouée quand le programme commence et libérée à la fin du programme). L'attribut *static* initialise la variable à 0 sauf si une autre valeur est spécifiée. Dans l'exemple ci-après, la variable *a* est allouée à chaque appel à la fonction, par contre *s*, qui est aussi une variable locale, est allouée une seule fois au début du programme. Ce qui signifie que le deuxième appel à la fonction réalloue la variable *a* et donc la réinitialise à 0, mais ne réalloue pas la variable *s* à cause de l'attribut *static*.

```
int g=0 ;//g est une variable gloable
void incrementation()
{
  int a=0 ; // variable locale à cette fonction
  static int s=0 ;// variable locale statique
  a++ ;
  s++ ;
  g++ ;
}
void main()
{
  incrementation() ;//a=1,s=1,g=1
  incrementation() ;//a=1,s=2,g=2
  g++ ;//g=3
  a++ ;//erreur
  s++ ;//erreur
}
```