

<b>1.</b>	<b>PRESENTATION GENERALE.....</b>	<b>3</b>
1.1	Historique: .....	3
1.2	Caractéristiques :.....	3
1.3	Objectifs atteints : .....	3
1.4	Remarque .....	3
1.5	Bibliographie .....	4
<b>2.</b>	<b>LES EXTENSIONS DE C (NON OBJETS).....</b>	<b>5</b>
2.1	Les commentaires .....	5
2.2	Conversions explicites .....	5
2.3	Utilisation des types enum, struct et union.....	5
2.4	Déclaration des variables .....	5
2.5	Fonction inline .....	6
2.6	Paramètres optionnels avec valeur par défaut .....	6
2.7	Surcharge des noms de fonctions .....	6
2.8	Opérateurs (Objets) de sortie et d'entrée: << et >> .....	7
2.9	Opérateurs : new et delete .....	7
2.10	fonction de gestion d'erreur d'allocation .....	8
2.11	Adressage par référence & .....	8
2.12	Les fonctions génériques (template) .....	9
<b>3.</b>	<b>INCOMPATIBILITES ENTRE C ET C++.....</b>	<b>10</b>
3.1	Entête de fonctions.....	10
3.2	Prototypes de fonctions.....	10
3.3	fonction sans argument .....	10
3.4	fonction sans valeur de retour .....	10
3.5	le type void *.....	10
3.6	Symbole de type const. ....	10

# Langage C++

**Pierre Bodin**

# 1. Présentation générale

## 1.1 Historique:

C++ a été développé vers 1980 par Bjarne Stroustrup (Bell Labs, AT & T )  
Améliorer le langage C

## 1.2 Caractéristiques :

- Le langage C++ se présente *presque* comme un *sur-ensemble* du langage C ANSI.
- Quelques incompatibilités entre C et C++.
- Le langage C++ contient des (améliorations) *caractéristiques nouvelles* par rapport au C qui ne sont pas spécifiquement *objet*.
- Le langage C++ offre la possibilité de programmation par *objets*.
- Rien n'impose au programmeur de programmer en utilisant les concepts de la POO.

## 1.3 Objectifs atteints :

- C++ conserve les aspects positifs de C:  
Portabilité, Concision, Efficacité, Bas niveau de langage
- Corrige les mauvais côtés de C : trop grande permissivité.
- **Permettre la programmation par OBJETS**

## 1.4 Remarque

C++ reste un langage relativement complexe.

## 1.5 Bibliographie

- Delannoy Claude **Programmer en langage C++** (248 F.) *Ed. Eyrolles*
- Delannoy Claude **Apprendre le C++ sous Turbo/Borland C++** (250 F.) *Ed. Eyrolles*
- Delannoy Claude **Exercices en langage C++. Programmation orientée objets** *Ed. Eyrolles*
- Leblanc Gérard **Turbo/Borland C++** (236 F.) (technique BIOS .... ) *Ed. Eyrolles*
- O'Reilley Cd, Oualline Steve **La programmation C++ par la pratique** (255 F.) (exemples)
- Meyer Jean Jacques **Borland C++ TurboC++ 3.0/3.1 pour Windows** *Ed. Dunod Tech*
- Weiskamp, Heiney K. L., Flamig B. **Object Oriented Programming with Turbo C++**  
*Ed. Wiley*
- Borland C++ 3.0 Guide du programmeur** Ed Borland
- Petzold Charles **Programmer sous Windows 3.1** *Ed. Microsoft Press.*
- B Stroustup. **Le langage C++ 2ième édition** *Ed. Addison-Wesley*
- Lipman S.B **L'essentiel du C++ 2ième édition** *Ed. Addison-Wesley*
- Meyer B. **Conception et programmation par objets pour du logiciel de qualité**  
*Ed. InterEditions*
- Charbonnel Jacquelin **Le langage C, les finesses d'un langage redoutable**
- Charbonnel Jacquelin **Langage C++, les spécifications du standard ANSI/ISO expliquées** (260 F.)  
InterEditions (2ième édition).
- Clavel G., Mirouze N., Pichon E., Soukal M. **Java la synthèse** *Ed. InterEditions*
- Fontaine Alain Bernard **La bibliothèque standard du C++** *Ed. InterEditions*

## 2. Les extensions de C (non objets)

### 2.1 Les commentaires

<b>C</b> /* commentaire en C */	<b>C ++</b> // commentaire en C++ et C ANSI  /* com. toujours acceptable en C++ */
------------------------------------	--

Un programme est beaucoup plus souvent lu qu'il n'est écrit

### 2.2 Conversions explicites

⇒ Nouvelle syntaxe : (le *cast* est toujours valable)

<b>C</b> int i = 1; float x = 3.2;  i = (int) x; x = (float) i;	<b>C ++</b> int i = 1; float x = 3.2;  i = int(x); x = float(i); Point p = Point(1.0, 2.0); (initialisation par l'appel d'une fonction constructive)  i = (int) x; // toujours valable x = (float) i;
--	--

### 2.3 Utilisation des types enum, struct et union

⇒ Le nom d'une énumération, d'une structure ou d'une union *est un type*.

<b>C</b> enum E { ..... }; struct S { ..... }; union U { ..... };  enum E e; struct S s, *ptr; Union U u;  Ptr = malloc( sizeof( struct S ) );	<b>C ++</b> enum E { ..... }; struct S { ..... }; union U { ..... };  E e; S s, *ptr; struct S s1; // toujours valable U u;  Ptr = malloc( sizeof( S ) );
<b>C</b> typedef enum { False, True } BOOL;  BOOL ouvert ;	<b>C ++</b> enum BOOL { False, True };  BOOL ouvert;

### 2.4 Déclaration des variables

⇒ On peut déclarer des variables en tout endroit dans une bloc et non plus uniquement avant la première instruction du bloc.

```

{
  int i;// déclaration de variables
  i = 2;    // instruction
  int j;// autre déclaration de variable
  j = i;
  ...
  for( int k = 0; k < 5; k++ ) {
    ...
  }
  // valeur de k ici ? (dépend des compilateurs !)
}

```

## 2.5 Fonction inline

⇒ Fonction expansée (développée) à chaque appel

```

inline int carre(int x) { return x*x; }

int j = carre( 2 );           // j vaut 4

```

Ressemble à une *macro C* mais faire très attention dans la macro (effet de bord) !

C	C ++
<code>#define ABS(x) (x)&gt;0 ? (x) : (-x)</code>	<code>inline int abs( int x ) { return x&gt;0 ? x : -x ;}</code>
<code>int i, k = -1;</code>	<code>int i, k = -1;</code>
<code>i = ABS( k ); // i vaut 1;</code>	<code>I = abs( k ); // i vaut 1;</code>

En C on utilise des macros expansés par le préprocesseur.

Inconvénient : parfois difficile à écrire, pas de contrôle de type, mise au point difficile.

## 2.6 Paramètres optionnels avec valeur par défaut

```

void f ( float, int = 2, char * = " " );
int g ( int *ptr = null , char ); // illégal

f ( 1.23, 10, " bonjour " );           // appel classique
f ( 1.23, 10 );                         // => f ( 1.23, 10, " " );
f ( 1.23 );                             // => f ( 1.23, 2, " " );
f ( )                                    // illégal
f ( 1.23, , " bonjour " );              // => illégal
f ( 1.23, " bonjour " );                // => illégal

```

⇒ La définition des valeurs par défaut peut se faire soit dans le prototype de la fonction, soit dans l'entête de sa définition, mais pas dans les 2 à la fois. Mettre de préférence dans le prototype.

⇒ Les valeurs par défaut sont placées à partir de la fin de la liste des paramètres.

## 2.7 Surcharge des noms de fonctions

*Surcharge* ou encore *surdéfinition* : plusieurs fonctions ont le même nom.

Les fonctions doivent être différenciables par leur *signature* (la nature et le nombre de leurs paramètres )

```
int max(int, int );
int max( const int * list );

// mais erreur car même signature et valeur retournée différente
char * cherche( char *);
int cherche( char * );
```

## 2.8 Opérateurs (Objets) de sortie et d'entrée: << et >>

Ces opérateurs sont en réalité des *opérateurs* relevant des possibilités *OBJET* du langage. Ils sont introduits ici pour pouvoir les utiliser tout de suite sans attendre tous les développements nécessaires (et longs) à leur explication.

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h> // pour pouvoir utiliser les flots cin et cout

int main()
{
    int n; float x;

    // méthode traditionnelle avec printf et scanf
    printf( "\n\nentrez un entier et un flotant : ");
    scanf( "%d %f", &n, &x );
    printf( " le produit de %5d par %10.2f est : %10.2f\n", n, x, n * x );

    // méthode C++ avec les flots cin et cout
    cout << "\n\nentrez un entier et un flotant : " ;
    cin >> n >> x ;
    cout << " le produit de " << n << " par " << x << " est " << n * x ;

    return 0;
}
```

<< est appelé *output* on insère dans le flot de sortie (opérateur d'insertion ou injection)

>> est appelé *input* on extrait du flot d'entrée (opérateur d'extraction)  
(comme un entonnoir !)

<<endl; // équivalant à /n mais plus performant en objet

4 flots prédéfinis :

```
cin      (équivalent stdin en C)
cout     (équivalent stdout en C)
cerr     (équivalent stderr en C)
clog
```

## 2.9 Opérateurs : new et delete

Utiliser les *opérateurs new et delete* de préférence à malloc et free.

syntaxe :

```
pointeur = new type ;           delete pointeur ;
pointeur = new type[expression] ; delete [] pointeur;
```

C	C ++
<pre>int *ptr, *tab;  ptr = (int *)malloc( sizeof( int ) ); tab = (int *)malloc( n * sizeof( int ) ); ..... free( tab ); free( ptr );</pre>	<pre>int *ptr, *tab;  ptr = new int; tab = new int[ n ]; ..... delete ptr; delete [ ] tab ;           // ou delete tab;</pre>

## 2.10 fonction de gestion d'erreur d'allocation

On peut définir une fonction qui sera appelée *automatiquement* si l'opérateur new échoue.

```
typedef void (*HANDLER)();
HANDLER HandlerPrecedent = set_new_handler( memoireSaturee );

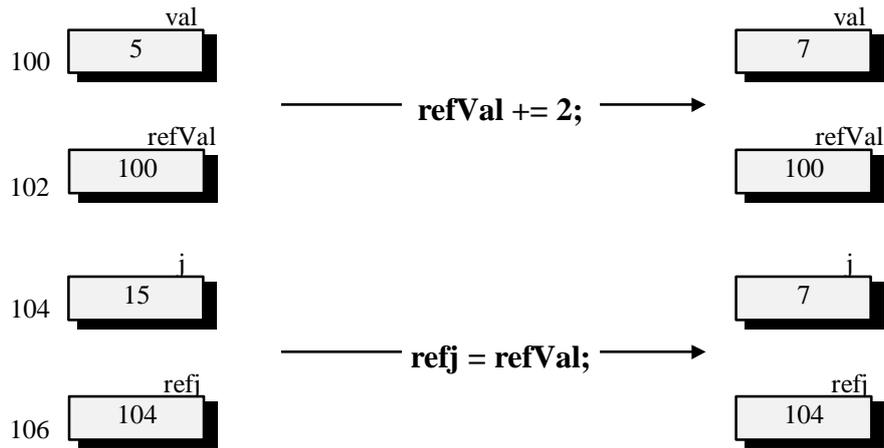
void memoireSaturee()
{
    cout << " memoire saturee " ;
    exit( 1 );
}

void f()
{
    HandlerPrecedent = set_new_handler( memoireSaturee );
    ....
    set_new_handler(HandlerPrecedent); // restauration
}
```

## 2.11 Adressage par référence &

<pre>int val = 10 int *pval = &amp;val // initialisation non obligatoire  *pval = 5 ; *pval += 2;</pre>	<pre>int val = 10; int &amp;refVal = val;// pas int &amp;refVal = &amp;val; // initialisation obligatoire  refVal = 5; // maintenant val vaut 5 refVal += 2; // maintenant val vaut 7 // identique à val += 2 ;</pre>
---	---

- ⇒ Nouveau type.
- ⇒ Si T est un type, T& est *le type référence vers T*
- ⇒ Toutes les manipulations de la référence s'effectuent sur l'objet référencé.
- ⇒ Une référence doit obligatoirement être initialisée lors de sa déclaration.
- ⇒ Une référence est un *alias* (un autre nom) pour la variable référencée.
- ⇒ On ne peut pas définir un pointeur sur une référence.



### 2.11.1 Utilisation dans les passages de paramètres

- ⇒ Equivalent au VAR du langage PASCAL
- ⇒ Syntaxe beaucoup plus lisible et naturelle.

C	C ++
<pre>void echange( int *x, int *y) {     int temp;     temp = *x; *x = *y; *y = temp; } void main() {     int a = 1, b = 2;     change( &amp;a, &amp;b ); }</pre>	<pre>void echange( int &amp;x, int &amp;y) {     int temp;     temp = x; x = y; y = temp; } void main() {     int a = 1, b = 2;     echange( a, b ); }</pre>

### 2.11.2 Retour d'une fonction par reference

C	C ++
<pre>int x, y ..... int *f( int b ) { return b ? &amp;x : &amp;y; } .... ... Possible mais trop complexe, à éviter !</pre>	<pre>int x, y ..... int &amp;f( int b ) { return b ? x : y; } .... int i = f( 1 ); // range x dans i f( 0 ) = 5; // range 5 dans y</pre>

on peut utiliser ainsi une fonction dans la partie gauche d'une expression

## 2.12 Les fonctions génériques (template)

création d'un modèle (d'un moule, d'un exemple) de fonction.  
 Le compilateur créera la fonction réelle quand ce sera utile.  
 Voir plus loin.

## 3. Incompatibilités entre C et C++

### 3.1 Entête de fonctions

La syntaxe Kernigham & Ritchie n'est plus acceptée

```
void echange( x, y)           // erreur de compilation
int &x, &y;
{ ...
}
```

### 3.2 Prototypes de fonctions

Le nombre et le type des paramètres de fonctions sont contrôlés à chaque appel.

Tout appel de fonction doit donc obligatoirement être précédé d'un *prototype* ou de la *définition* de la fonction. (Comme en C ANSI)

### 3.3 fonction sans argument

```
float f1 ( ) ;           // fonction sans argument et pas float f ( void);
```

### 3.4 fonction sans valeur de retour

```
void f2 ( ) ;           // ni f2 ( ) ni void f2( void ) ;
```

### 3.5 le type void \*

la conversion *implicite* n'est possible que dans le sens  $\text{void} * \leftarrow \text{Type} *$

```
void *generic;
int *ptr;

generic = ptr;           // ok
ptr = generic;          // erreur à la compilation
ptr = ( int *)generic;  // ok
```

### 3.6 Symbole de type const.

Le mot clé *const*, placé devant un symbole, est un modificateur qui signifie que la valeur du symbole ne doit pas être modifiée.

- En C++ un symbole défini avec *const* est local au fichier : on peut redéfinir le même symbole dans un autre fichier (avec une valeur différente)

en C il y a une erreur à l'édition de liens.

- en C++, *const* est utilisé en remplacement de la directive *#define* du préprocesseur.

En C	en C++
<pre>#define MAX 100 char tab[ MAX + 1]</pre>	<pre>const int max = 100; char tab[ MAX + 1]</pre>

- En C++ une constante doit obligatoirement être initialisée lors de sa déclaration.