

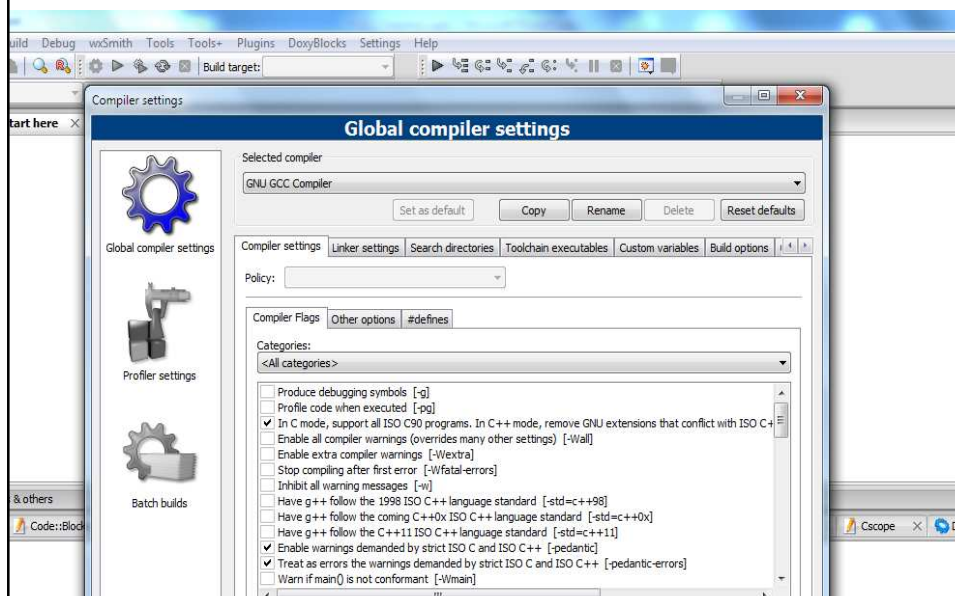
Plan

- ❑ Bref historique
- ❑ Spécificités du C++ : synthèse
- ❑ Commentaires
- ❑ Types de données – Conversions
- ❑ Déclarations et initialisations – Constantes
- ❑ Entrées/Sorties
- ❑ Fonctions – Paramètres – Surcharge – Fonctions inline
- ❑ Références – passage de paramètres par référence
- ❑ Gestion dynamique de la mémoire
- ❑ Espaces de nommage

Bref historique

- ❑ 1980 : création du langage C++ par Bjarne Stroustrup dans les laboratoires Bell d'AT&T.
- ❑ 1985 : parution de la première édition du livre « *The C++ programming Language* »
- ❑ 1995-1997 : révision standard ANSI/ISO

Compiler settings 'Cde::Blocks'



Remarque!

Les notations introduites par C
restent valables pour le C++



Nouvelles possibilités pour palier aux carences du C

Spécificités du C++ : synthèse

C++ présente, par rapport au C, des extensions :

- ▣ qui permettent de supporter la POO.
- ▣ qui ne sont pas orientées POO (contenu de ce chapitre) :
 - ▣ Nouvelle forme de commentaires.
 - ▣ Type booléen + définition implicite de nouveaux types de données.
 - ▣ Plus grande souplesse dans les déclarations et les initialisations.
 - ▣ Nouvelles possibilités d'entrées/sorties.
 - ▣ Surcharge des fonctions : attribution d'un même nom à différentes fonctions.
 - ▣ Notions de référence - passage d'arguments par référence.
 - ▣ Possibilité de définir des fonctions "en ligne" (inline).
 - ▣ Nouveaux opérateurs de gestion dynamique de la mémoire : new et delete.
 - ▣ Résolution des conflits de noms grâce aux espaces de nommage.

Commentaires

- Commentaires en C :

/ Commentaire en C pouvant s'étendre sur
plusieurs lignes */*

- Commentaires en C++ :

- Commentaires de type C (pour désactiver une partie du code) :

/ Commentaire en C++ pouvant s'étendre sur
plusieurs lignes */*

- Nouveau type de commentaires (commentaire de fin de ligne) :

// Commentaire en C++ sur la même ligne

le commentaire de type // n'appartient qu'au C++

Commentaires

- Nous pouvons mêler (volontairement ou non !) les deux types de commentaires.

- Dans l'exemple suivant :

/ partie1 // partie2 */* partie3

Le commentaire ouvert par /* ne se termine qu'au prochain */ donc partie1 et partie2 sont des commentaires, tandis que partie3 est considérée comme appartenant aux instructions.

- De même, dans :

partie1 *// partie2 /* partie3 */* partie4

Le commentaire introduit par // s'étend jusqu'à la fin de la ligne. Il concerne donc partie2, partie3 et partie 4.

Types de données

- Le langage C ne possède pas de type booléen.
- C++ a introduit un nouveau type **bool** pour pallier cette carence.
- Le type **bool** est formé de deux valeurs notées **true** et **false**.

```
bool flag = true;
...
do
{
    ...
    if ( ... )
        flag = false;
    ...
} while (flag == true);
```

Types de données

- En théorie, les résultats des comparaisons ou des combinaisons logiques doivent être de type **bool**. Toutefois, il existe des conversions implicites :
 - **bool** → type numérique : **true** devient 1, **false** devient 0.
 - type numérique → **bool** : toute valeur $\neq 0$ devient **true** et 0 devient **false**.
- Dans ces conditions, tout se passe finalement comme si **bool** était un type énuméré défini ainsi :

```
typedef enum { false=0, true } bool;
```

Types de données

- Nouvelle possibilité en C++ : Définition implicite de types sans avoir à utiliser le mot clé **typedef**
 - ```
struct coordonnees {
 int x;
 int y;
};
```
  - ```
enum couleur = {rouge, vert, bleu};
```
 - ```
union numerique {
 int entier;
 double reel;
};
```
- **coordonnees**, **couleur** et **numerique** sont considérés comme types de données et peuvent être utilisés en tant que tels.

## Conversions de types

- C et C++ autorisent le mélange de données de différents types dans une même expression.
- Nécessité de convertir toutes les données dans un type commun avant d'évaluer l'expression.
- 2 types de conversions :
  - Conversions implicites (automatique) : mises en place automatiquement par le compilateur en fonction du contexte sans l'intervention du programmeur.
  - Conversions explicites (**cast**) : mises en place par le programmeur qui choisit explicitement le type dans lequel les données doivent être converties.

## Conversions implicites

- Le compilateur tente généralement de convertir les données du type le plus 'petit' vers le type le plus 'large' → **conversion non dégradante**

```
float X = 12.3;
int N = 5;
X = X + N; //la valeur de N est convertie en float (5.0)
```

- Dans le cas d'une affectation, le résultat de l'évaluation de l'expression à droite, est converti dans le type de la variable à gauche. Il peut y avoir perte de précision si le type de la destination est plus faible que celui de la source → **conversion dégradante**

```
float X = 12.3;
int N;
N = X; //la valeur de X est convertie en int (12)
```

## Conversions implicites

- Exemple:

```
int X;
float A = 12.48;
char B = 4;
X = A / B;
```

Le résultat de la division est de type `float` (valeur 3.12)

Il sera converti en `int` avant d'être affecté à `x` (conversion dégradante), ce qui conduit au résultat `x=3`.

## Conversions explicites

- Le programmeur peut recourir à une conversion explicite (ou **cast**) pour forcer la conversion dans un type particulier.
- 2 manières d'effectuer une conversion explicite en C++ :
  - Ancienne notation introduite par C  
`(type) expression`
  - Nouvelle notation fonctionnelle introduite par C++  
`type (expression)`
- Possibilité de mixage des deux notations  
`(type) (expression)`

## Conversions explicites

- Exemples :

```
int A = 3, B = 4;
float C;
C = (float) A / B;
```

La valeur de `A` est explicitement convertie en `float`. Le résultat de la division (type rationnel, valeur 0.75) est affecté à `C`.  
Résultat: `C=0.75`

```
double d = 2.5;
int i;
i = int(d); // i = 2 (conversion dégradante)
```



## Conversions et pointeurs

- ```
void * gen ;
int * adi ;
```
- ces deux affectations sont légales en C

```
gen = adi ;
adi = gen ;
```
 - Elles font intervenir des "conversions implicites":
 - `int * -> void *` pour la première,
 - `void * -> int *` pour la seconde.
 - En C++, seule la **conversion d'un pointeur quelconque en void *** peut être implicite.
 - Avec les déclarations précédentes, seule l'affectation : `gen = adi;` est acceptée.
 - il reste possible de faire appel explicitement à la conversion
 - `adi = (int *) gen`
- La conversion d'un pointeur de type quelconque en void * revient à ne s'intéresser qu'à l'adresse correspondante au pointeur, en ignorant son type.
 - La conversion inverse de void * en un pointeur de type donné revient à associer un type à une adresse.
 - Cette seconde possibilité est plus dangereuse que la première ; elle peut même obliger le compilateur à introduire des modifications de l'adresse de départ, dans le seul but de respecter certaines contraintes d'alignement (liées au type d'arrivée). C'est la raison pour laquelle cette conversion ne fait plus partie des conversions implicites en C++.

Exercice 1

Soient les déclarations :

```
char c = 'A' ;
int n = 5 ;
long p = 1000 ;
float x = 1.25 ;
double z = 5.5 ;
```

Quels sont le type et la valeur de chacune des expressions suivantes :

```
n + c + p          /* 1 */
2 * x + c          /* 2 */
(char) n + c       /* 3 */
(float) z + n / 2  /* 4 */
```

Exercice 2

Soient les déclarations :

```
int n = 5, p = 9, int q ;
float x ;
```

Quels sont le type et la valeur de chacune des expressions suivantes :

```
q = n < p ;           /* 1 */
q = n == p ;          /* 2 */
q = p % n + p > n ;   /* 3 */
x = p / n ;           /* 4 */
x = (float) p / n ;    /* 5 */
x = (p + 0.5) / n ;    /* 6 */
x = (int) (p + 0.5) / n ; /* 7 */
q = n * (p > n ? n : p) ; /* 8 */
q = n * (p < n ? n : p) ; /* 9 */
```

Pluralité	Opérateurs
Binaire	() ⁽³⁾ [] ⁽³⁾ .> ⁽¹⁾⁽²⁾ (3)
Unaire	+ - ++ ⁽⁵⁾ -- ⁽⁵⁾ ! ~ * & ⁽¹⁾ new ⁽¹⁾⁽⁴⁾⁽⁶⁾ new[] ⁽¹⁾⁽⁴⁾⁽⁶⁾ delete ⁽¹⁾⁽⁴⁾⁽⁶⁾ delete[] ⁽¹⁾⁽⁴⁾⁽⁶⁾ (cast)
Binaire	* / %
Binaire	*.> ⁽¹⁾ .* ⁽¹⁾
Binaire	+ -
Binaire	<< >>
Binaire	< <= > >=
Binaire	== !=
Binaire	&
Binaire	^
Binaire	
Binaire	&&
Binaire	
Binaire	= ⁽¹⁾⁽³⁾ += -= *= /= %= &= ^= = <<= >>=

Déclarations et initialisations

- ❑ C++ offre une plus grande souplesse en matière de déclarations que le C.
- ❑ Il n'est plus obligatoire de regrouper les déclarations au début d'un bloc ou d'une fonction comme en C.
- ❑ En C++, il est possible de déclarer des variables à n'importe quel endroit du code, à condition que ce soit fait avant leur utilisation.
- ❑ La portée de la variable reste limitée au bloc ou à la fonction dans laquelle elle a été déclarée.

Déclarations et initialisations

■ C

```
int a=5, b=2, n;  
float x;  
...  
n = a / b;  
x = a / (float) b;  
...
```

Séparation entre les déclarations qui se trouvent au début du code et les instructions qui viennent après.

Déclaration tardive d'une variable au milieu des instructions : plus grande liberté d'emplacement des déclarations.

■ C++

```
int a=5, b=2, n;  
...  
n = a / b;  
...  
float x;  
x = a / (float) b;  
...
```

Déclarations et initialisations

- L'instruction suivante est acceptée en C++ mais pas en C :

```
for (int i=0 ; i<N ; i++)  
{  
    x += i; // i variable locale de parcours  
}
```

- Cette possibilité s'applique à toutes les instructions structurées, c'est-à-dire aux instructions **for**, **switch**, **while** et **do...while**.
- Dans ce cas la portée de la variable **i** est limitée au bloc régi par la boucle **for** : en dehors de ce bloc elle ne sera pas reconnue !

Déclarations et initialisations

- En C++ les expressions utilisées pour initialiser une variable peuvent être quelconques, alors qu'en C elles ne peuvent faire intervenir que des variables dont la valeur est connue dès l'entrée dans la fonction concernée.
- Voici un exemple incorrect en C et accepté en C++ :

```
main()  
{  
    int n ;  
    ...  
    n = ...  
    ...  
    int q = 2*n - 1 ;  
    ...  
}
```

Constantes

- En C il existe deux types de constantes :

- **Constantes symboliques** : constantes typées définies grâce au mot clé **const**. La valeur d'une constante symbolique n'est pas connue au moment de la compilation.

```
const int n = 10;  
int tab[n]; /* interdit en C */
```

- **Constantes nommées** : constantes non typées définies grâce à la directive du préprocesseur **#define**. La valeur d'une constante nommée est connue au moment de la compilation.

```
#define N 10  
...  
int tab[N];  
char ch[2*N];  
/* autorisé car la valeur de N est connue à ce niveau */
```

Constantes

- En C++ le mot clé **const** permet de définir des constantes typées dont la valeur est connue au moment de la compilation (permet de remplacer la directive du préprocesseur **#define**)

```
const int n = 10;  
int tab[n];    // interdit en C et autorisé en C++  
char ch[2*n]; // interdit en C et autorisé en C++
```

- Exemples

```
const float PI = 3.14;  
const int MAX = 100;  
const char NOM[] = "etudiant II1";
```

Constantes

□ Utilisation de **const** avec des pointeurs

```
int n=8, m=5, p=3;
const int * N = &n; //pointeur sur entier constant
int * const M = &m; //pointeur constant sur entier
const int * const P = &p;
//pointeur constant sur entier constant
```

□ De manière générale

```
const T * p = &v;
// pointeur modifiable p vers un objet non modifiable de type T
T * const p = &v;
// pointeur p non modifiable vers un objet modifiable de type T
const T * const p = &v;
// pointeur p non modifiable vers un objet non modifiable de
// type T
```

Constantes

□ Exemples

□ Donnée pointée constante

```
const char * ptr = "hello";
*ptr = 'H'; // Erreur (assignment to read-only location)
ptr++; // Correct
```


□ Pointeur constant

```
char * const ptr = "hello";
*ptr = 'H'; // Correct
ptr++; // Erreur (increment of read-only variable)
```

□ Donnée pointée et pointeur constants

```
const char * const ptr = "hello";
*ptr = 'H'; // Erreur (assignment to read-only location)
ptr++; // Erreur (increment of read-only variable)
```

Entrées / Sorties

- En C++, les fonctions d'entrée/sortie standard en C telles que **printf()** et **scanf()** déclarées dans le fichier d'en-tête **<stdio.h>** restent utilisables.
- Nouvelles possibilités d'entrées/sorties en C++ reposant sur la notion de flot (*stream*).
- Ces nouvelles possibilités ne nécessitant pas le **FORMATAGE** des données !

- La bibliothèque standard **<iostream.h>** du langage C++ fournit des opérations qui peuvent se substituer à celles de la bibliothèque **<stdio.h>** du langage C.

Entrées / Sorties

- Flots standards

	C	C++
Entrées	stdin	cin
Sorties	stdout	cout
Erreurs	stderr	cerr

- La sortie de données sur les flots de sortie (**cout** et **cerr**) se fait en utilisant l'opérateur de redirection **<<**
- L'entrée de données à partir du flot d'entrée (**cin**) se fait en utilisant l'opérateur de redirection **>>**

Entrées / Sorties

- Ecrire des données avec **cout**

```
#include<iostream.h>
...
int n = 25;
cout << "valeur de n : " ;
cout << n;
```

Ces 2 dernières instructions permettent d'afficher le résultat suivant :

valeur de n : 25

Elles peuvent être remplacées par l'instruction suivante :

```
cout << "valeur de n : " << n;
```

Entrées / Sorties

- **Cout** permet d'écrire des données de tout type sans formatage préalable :

```
int n = 25; char c = 'a'; float x = 12.345;
char * ch = "bonjour"; int * ad = &n;
cout << "valeur de n : " << n << '\n';
cout << "valeur de n^2 : " << n*n << '\n';
cout << "caractere c : " << c << "\n";
cout << "valeur de x : " << x << "\n";
cout << "chaine ch : " << ch << endl;
cout << "adresse de n : " << ad << endl;
```

- Pour le retour à la ligne nous pouvons utiliser soit le caractère **'\n'**, soit la chaîne **"\n"** ou simplement l'objet **endl**.

Entrées / Sorties

- Lire des données avec **cin**

```
#include<iostream.h>
...
int id; char nom[20]; float moyenne;
cout << "saisir l'identifiant, le nom et la moyenne" ;
cin >> id;
cin >> nom;
cin >> moyenne;
```

Ces 3 dernières instructions peuvent être remplacées par l'instruction suivante :

```
cin >> id >> nom >> moyenne;
```

Fonctions

- Mêmes règles de définition qu'en C

```
type_ret nom_fnct(type_1 arg_1, ..., type_n arg_n)
{
    // déclarations locales
    // instructions
}
```

- Typage obligatoire des fonctions : toute fonction doit avoir un type de retour, si elle ne retourne rien le type doit obligatoirement être **void**.
- Déclaration obligatoire des fonctions grâce aux prototypes si nécessaire.

Appel de fonction et conversion de type

- Le C++, contrairement au C, autorise dans une certaine mesure le non-respect du type des arguments lors d'un appel de fonction : le compilateur effectue alors une conversion de type.

```
double ma_fonction(int u, float f);  
...  
int main()  
{  
    char c; int i, j; float x; double r1, r2, r3, r4;  
    r1 = ma_fonction(i,x); // appel standard  
    r2 = ma_fonction(c,x); // correct, c est converti en int  
    r3 = ma_fonction(i,j); // correct, j est converti en float  
    r4 = ma_fonction(x,j); // correct, x est converti en int  
                           // et j est converti en float  
}
```

Surdéfinition/surcharge des fonctions

- Surdéfinition/surcharge d'une fonction : possibilité d'attribuer le même nom à plusieurs fonctions pouvant effectuer des traitements différents (avec des conditions). Ceci est interdit en C, mais autorisé en C++.
- Lors de l'appel, le compilateur identifie la bonne fonction grâce à sa signature (**nombre et types des arguments**). Le type de retour ne fait pas partie de la signature, il n'est donc pas pris en compte dans l'identification de la fonction.
- Exemple :

```
int fonction(int n) { ... }  
int fonction(float x) { ... }  
void fonction(int n, int p) { ... }  
float fonction(float x, int n) { ... }  
void fonction(float x, int n) { ... } // erreur : même signature
```

Règles de recherche d'une fonction surchargée

- Le compilateur recherche la "meilleure correspondance" possible. Il y a plusieurs niveaux de correspondance :
 - 1) Correspondance exacte : le type de l'argument passé correspond exactement au type du paramètre formel.
 - 2) Correspondance avec promotion numérique : c.à.d. avec un recours à une conversion de types, essentiellement char et short → int et float → double
 - 3) Conversions standards : il s'agit des conversions légales en C++ ; en général toute conversion d'un type numérique en un autre type numérique.
- Si plusieurs fonctions conviennent, il y a erreur de compilation due à l'ambiguïté. De même, si aucune fonction ne convient à aucun niveau, il y a aussi erreur de compilation.

Exercice

Soient les déclarations suivantes :

```
int fct (int) ;           // fonction I
int fct (float) ;        // fonction II
void fct (int, float) ;   // fonction III
void fct (float, int) ;   // fonction IV

int n, p ; float x, y ; char c ; double z ;
```

Les appels suivants sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions éventuellement mises en place ?

- | | |
|------------------------------|------------------------------|
| a. <code>fct (n) ;</code> | f. <code>fct (n, p) ;</code> |
| b. <code>fct (x) ;</code> | g. <code>fct (n, c) ;</code> |
| c. <code>fct (n, x) ;</code> | h. <code>fct (n, z) ;</code> |
| d. <code>fct (x, n) ;</code> | i. <code>fct (z, z) ;</code> |
| e. <code>fct (c) ;</code> | |

Paramètres avec valeurs par défaut

- Il arrive parfois qu'on soit amené à passer toujours la même valeur en paramètre à une fonction.
- C++ offre la possibilité de définir des valeurs par défaut pour les paramètres d'une fonction lors de sa déclaration (dans le prototype).
- Restriction importante : les paramètres ayant une valeur par défaut doivent être les derniers de la liste des paramètres c.à.d. déclarés le plus à droite.
- Exemple :

```
int fonction_1(int n, int p=5);  
void fonction_2(float x, int n, char c='A', int p=2);
```

Paramètres avec valeurs par défaut

- Exemple :

```
void f(int n, int p=1, float x=2.5);  
...  
int main()  
{  
    ...  
    f(a);           // correct ~ f(a, 1, 2.5)  
    f(a, b);        // correct ~ f(a, b, 2.5)  
    f(a, b, c);     // correct : présence des 3 paramètres  
    ...  
}
```

Paramètres avec valeurs par défaut

- Pourquoi faut-il respecter la contiguïté ?

```
void f(int p=1, int n, float x=2.5);  
...  
int main()  
{  
    ...  
    f(a, b, c);    // présence des 3 paramètres  
    f(a);          // f(1, a, 2.5)  
    f(a, b);       // problème !  
                  // f(a, b, 2.5) ou f(1, a, b) ?  
    ...  
}
```

Paramètres anonymes

- En C le nommage des paramètres d'une fonction est obligatoire.

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

- En C++, il est possible de déclarer des paramètres anonymes dans la signature d'une fonction.
- Paramètre anonyme : paramètre dont seul le type est spécifié.

```
int main(int, char**)  
{  
    //paramètres anonymes non utilisés  
}
```

Références/Alias

- En C, pour référencer une variable nous utilisons des pointeurs :

```
int n ;  
int* p = &n ; // p est une référence à n
```

- En C++, une référence (ou alias) est un identificateur supplémentaire pour une variable (deuxième nom).
- Pour créer une référence en C++, on utilise l'opérateur de référence & :

```
int n ;  
int& r = n ; // r est une référence à n
```

Références/Alias

- Physiquement, une référence est un pointeur constant sur la variable référencée, mais qui est manipulée par le programme comme s'il s'agissait de la variable elle-même.

```
int n ;  
int& r = n ; // r et n désignent la même variable  
n = 3 ;  
cout << r ; // affiche 3  
r = 5 ;  
cout << n ; // affiche 5
```

- En C++, toute opération citant une référence porte sur la variable référencée et non sur le pointeur.

Références/Alias

- Une référence ne peut se référer qu'à une seule variable pendant sa durée de vie :

```
int n, p=9 ;
int& r = n ;
r = p ; /* signifie que la valeur de r et donc celle
de n reçoit la valeur de p et ne signifie pas que r
devient une référence sur p ! */
```

- Une référence doit obligatoirement être initialisée au moment de sa déclaration, elle ne peut pas être initialisée avec une valeur constante ni avec une expression :

```
int& r ; // incorrect
int& r = 5 ; // incorrect
int& r = 2*n ; // incorrect
```

Références/Alias

- Une référence ne peut se référer qu'à une seule variable pendant sa durée de vie :

```
int i=5, j=6;
int& k = i; // k variable référence sur i
cout << i << " " << j << " " << k << endl; // affiche 5 6 5
k=3;
cout << i << " " << j << " " << k << endl; // affiche 3 6 3
i=8;
cout << i << " " << j << " " << k << endl; // affiche 8 6 8
k=j;
cout << i << " " << j << " " << k << endl; // affiche 6 6 6
k=1;
cout << i << " " << j << " " << k << endl; // affiche 1 6 1
```

Passage de paramètres

- En C, le passage de paramètres à une fonction se fait uniquement par valeurs.
- Le passage par valeurs ne permet pas de modifier les paramètres effectifs.
- Pour simuler un passage par adresses on est obligé de travailler avec les pointeurs (le passage se fait toujours par valeurs mais dans ce cas il s'agit de la valeur d'un pointeur c.à.d. une adresse)
- C++ pallie cette lacune en introduisant le **passage de paramètres par références**

Passage par valeurs (rappel)

```
void fonction(double a)
{
    a = 3;
}
```

```
void main(void)
{
    double x = 1;
    fonction(x);
    cout << x;
}
```

→ x vaut 1

En mémoire :

A l'appel de
fonction(x);



Dans fonction:
a = 3;

x: double
1

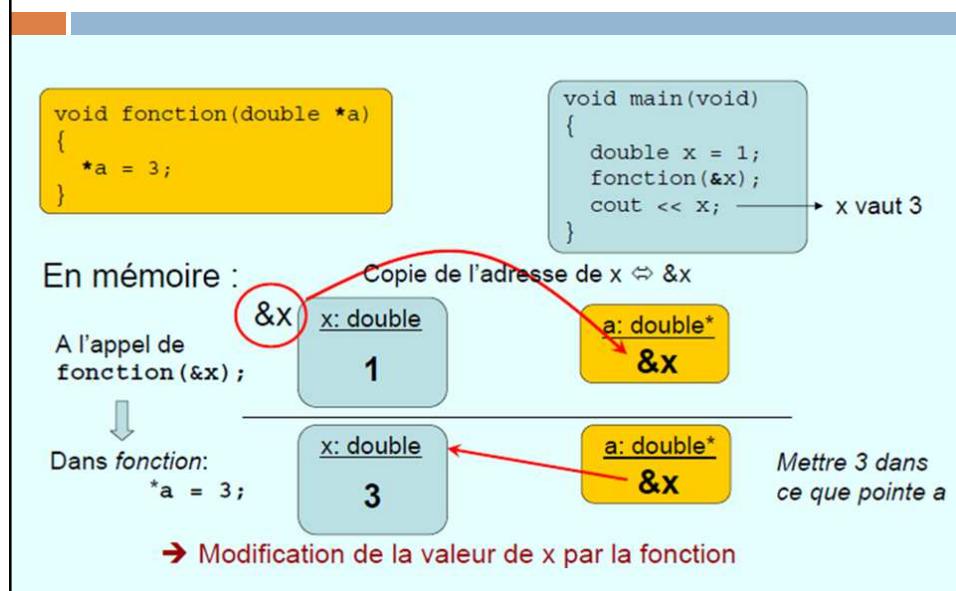
Copie des valeurs

a: double
1

a: double
3

→ Pas de modification de la valeur de x dans la fonction

Passage par adresses (rappel)



Passage par références

```
void increment (int& n, int i)
{ n = n + i ; }
...
int main()
{ int x = 7 ;
  increment(x, 3) ;
  cout << x << endl ; // affiche 10
}
```

- La notation `int& n` signifie que le paramètre `n` de type `int` est passé par référence.
- Le compilateur se charge d'extraire l'adresse du paramètre : le passage se fait par adresse mais de manière implicite.
- Dans le corps de la fonction on manipule le paramètre `n` lui même et non un pointeur sur ce paramètre. De même au moment de l'appel on passe une valeur (variable) et non une adresse (pointeur).

Passage par références

```
void fonction(double &a)
{
    a = 3;
}
```

```
void main(void)
{
    double x = 1;
    fonction(x);
    cout << x;
}
```

→ x vaut 3

En mémoire :

A l'appel de
fonction(x);

Dans fonction:
a = 3;

&x

x: double

1

a: double&

&x

a est une
référence de x

x: double

3

a: double&

&x

Mettre 3 dans ce
que référence a

→ Modification de la valeur de x par la fonction

Passage par références

Version C

```
void permut (int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
...
int i=2, j=4;
permut (&i, &j);
/* i vaut 4 et j vaut 2 */
```

Version C++

```
void permut (int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
...
int i=2, j=4;
permut (i, j);
/* i vaut 4 et j vaut 2 */
```

Propriétés du passage par référence

❑ Absence de conversion

Dès lors qu'une fonction prévoit un passage de paramètres par référence, les conversions de type à l'appel ne sont plus autorisées : le type du paramètre effectif (réel) doit correspondre exactement au type du paramètre formel (fictif).

```
void f(int& n) ; // f reçoit un entier par référence
float x ;
...
f(x) ; // incorrect : discordance de types
```

Exception à cette règle : cas des paramètres formels constants !

Propriétés du passage par référence

❑ Cas d'un paramètre effectif constant

Dès lors qu'une fonction prévoit un passage de paramètres par référence, elle n'admettra pas de constantes comme paramètres effectifs au moment de son appel.

```
void f(int& n) ; // f reçoit un argument par référence
...
f(3) ; // incorrect : f ne peut pas modifier une constante
const int c = 5;
f(c) ; // incorrect : f ne peut pas modifier une constante
```

Exception à cette règle : cas des paramètres formels constants !

Propriétés du passage par référence

□ Cas d'un paramètre formel constant

Considérons une fonction ayant le prototype suivant :

```
void f(const int& n) ;
```

Les appels suivants sont corrects :

```
const int c = 5 ; float x;  
f(3) ; // correct  
f(c) ; // correct  
f(x) ; /* correct : f reçoit par référence une variable  
temporaire contenant le résultat de la conversion de x en  
int */
```

Ce cas est équivalent à un passage de paramètres par valeur sans copie des paramètres effectifs dans les paramètres formels.

Retour d'une référence

- Le passage des arguments par référence s'applique aussi à la valeur de retour d'une fonction.

→ Il est possible qu'une fonction en C++ retourne une référence.

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

- Un appel de **f** provoquera la transmission en retour non plus d'une valeur, mais de la référence de **n**. Cependant, on peut utiliser **f** d'une façon usuelle :

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

Fonctions en ligne (inline)

Les macros en C

- Macro : notion voisine d'une fonction (petite fonction)
- Une macro et une fonction s'utilisent apparemment de la même façon, en faisant suivre leur nom d'une liste d'arguments entre parenthèses.
- Elle permet de factoriser un morceau de code peu volumineux sans avoir recours aux fonctions
- Une macro est définie en utilisant la directive **#define** du préprocesseur
#define MAX(a,b) ((a)>=(b))?(a):(b)
- Une macro est incorporée dans le code par le préprocesseur à chaque fois qu'elle est appelée (remplacement textuel),

Macro vs Fonction

- les instructions correspondant à une macro sont incorporées au programme à chaque appel: l'incorporation est réalisée au niveau **du préprocesseur**.
- les instructions correspondant à une fonction sont "générées" une seule fois **par le compilateur** sous forme d'instructions en langage machine; à chaque appel, il y aura seulement mise en place des instructions nécessaires pour établir la liaison entre le programme et la fonction : sauvegarde de l'état courant , recopie des valeurs des arguments, recopie de la valeur de retour, restauration de l'état courant et retour dans le programme
→ perte de temps d'exécution en comparaison aux macros

Fonctions en ligne (inline)

Les macros en C

- ❑ L'utilisation des macros engendre du code volumineux et donc une perte d'espace mémoire d'autant plus importante que le code correspondant à la macro est important.
- ❑ L'utilisation des macros permet de gagner en temps d'exécution puisqu'il n'y a pas de mécanisme d'appel comme dans les fonctions.
- ❑ Les macros présentent un risque conséquent d'effet de bord lié à leur mauvaise définition et/ou utilisation :

```
#define CARRE(a) ((a)*(a))
```

CARRE(n++) sera remplacée par **(n++)*(n++)** et donc n sera incrémentée 2 fois !

Fonctions en ligne (inline)

- ❑ C++ nous offre la possibilité de tirer profit des avantages des macros et de ceux des fonctions en définissant les **fonctions en ligne** (ou **inline**)
- ❑ Une fonction en ligne se définit et s'utilise comme une fonction ordinaire, à la seule différence qu'on fait précéder son en-tête du mot clé **inline**

```
inline int max(int a, int b) { return (a>=b)?a:b; }
```
- ❑ Lorsqu'une fonction est déclarée inline, le compilateur la traite comme s'il s'agissait d'une macro (remplacement de code) dans la mesure du possible (fonction peu volumineuse).
- ❑ Les fonctions inline permettent d'avoir la sécurité d'une fonction ordinaire (pas d'effet de bord) et la rapidité d'une macro.
- ❑ Une fonction en ligne doit être définie dans le même fichier source que celui où on l'utilise.

Les fonctions inline: code plus rapide ou plus lent?

- **Les fonctions inline peuvent rendre le code plus rapide:** l'intégration du code peut supprimer une poignée d'instructions inutiles, ce qui peut rendre le code plus rapide.
- **Les fonctions inline peuvent rendre le code plus lent:** un excès de fonctions inline peut rendre le code 'indigeste', ce qui peut provoquer un excès d'accès à la mémoire virtuelle sur certains systèmes. En d'autres mots, si la taille de l'exécutable est trop importante, le système risque de passer beaucoup de temps à faire de la pagination sur disque pour accéder à la suite du code.

Gestion dynamique de la mémoire

- En C, l'allocation et libération de la mémoire se font grâce aux fonctions `malloc()` et `free()`. Ces fonctions restent valables en C++.
- C++ a introduit de nouveaux opérateurs pour l'allocation et la libération de la mémoire : `new` et `delete`.

Gestion dynamique de la mémoire

- L'opérateur **new** s'utilise ainsi pour créer des variables simples :

```
new type
```

où **type** représente un type quelconque. Il fournit comme résultat un pointeur (de type **type***) sur l'emplacement mémoire correspondant.

- L'opérateur **new** s'utilise aussi pour créer des tableaux :

```
new type[n]
```

où **n** désigne une expression entière quelconque (non négative). Cette instruction alloue l'emplacement nécessaire pour **n** éléments du type indiqué; elle fournit en résultat un pointeur (de type **type***) sur le premier élément de ce tableau.

Gestion dynamique de la mémoire

- Lorsqu'on souhaite libérer un emplacement préalablement alloué par **new**, on doit utiliser l'opérateur **delete**.

```
delete ptr;
```

Cette instruction permet de libérer l'emplacement alloué à une variable simple et dont l'adresse se trouve dans le pointeur **ptr**.

- Lorsqu'on souhaite libérer un emplacement préalablement alloué par **new .. []**, on doit utiliser l'opérateur **delete[]**.

```
delete[] tab;
```

Cette instruction permet de libérer l'emplacement alloué à un tableau et dont l'adresse se trouve dans le pointeur **tab**.

Gestion dynamique de la mémoire

malloc() / free()

```
//Allocation d'une variable simple
int* ptr;
ptr = (int*) malloc(sizeof(int));
...
//Libération d'une variable simple
free(ptr);

//Allocation d'un tableau
int n = N_MAX;
int* tab;
tab = (int*) malloc(n*sizeof(int));
...
//Libération d'un tableau
free(tab);
```

new / delete

```
//Allocation d'une variable simple
int* ptr;
ptr = new int;
...
//Libération d'une variable simple
delete ptr;

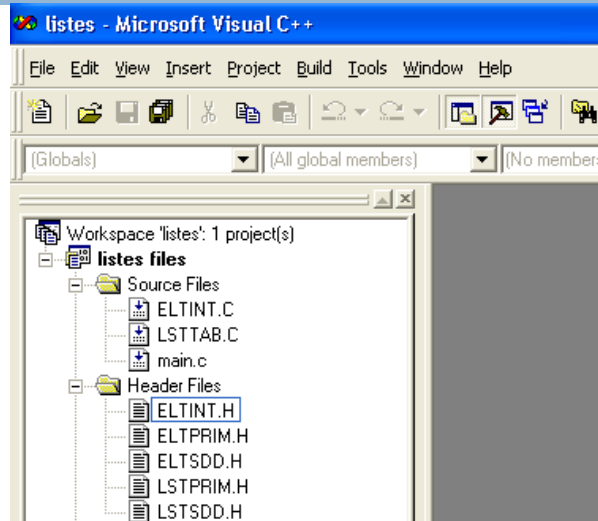
//Allocation d'un tableau
int n = N_MAX;
int* tab;
tab = new int[n];
...
//Libération d'un tableau
delete[] tab;
```

Structure d'un programme C++

La structure d'un programme C++ est très similaire à la structure d'un programme C. On y retrouve :

- ❑ une fonction principale : main() c'est le point d'entrée au programme.
- ❑ Les fichiers d'en-tête (extension .h) regroupent les déclarations de types et de classes, les prototypes des fonctions...
- ❑ Les fichiers sources (extensions : .c, .cpp) comportent le code proprement dit : le corps des fonctions et des méthodes, la déclaration de variables, d'objets.

Exemple: déclaration d'un TDA



TDA ELEMENT

- Tous les éléments appartiennent à un TDA ELEMENT qui possède un élément vide et sur lequel on peut :
 - saisir un élément
 - afficher un élément
 - affecter un élément dans un autre élément
 - copier les informations d'un élément dans un autre élément (le dupliquer)
 - comparer deux éléments
 - allouer dynamiquement la mémoire ou initialiser un élément
 - libérer la mémoire allouée pour un élément
- On va créer un header (**ELTPRIM.H**)

TDA ELEMENT

```
/* *****  
* Fichier : ELTPRIM.H  
* Contenu : Déclaration des primitives du TDA ELEMENT.  
* ***** */  
  
#ifndef _ELTPRIM_H  
#define _ELTPRIM_H  
#include "ELTSDD.H"  
  
/* Lecture d'un élément */  
void elementLire(ELEMENT *);  
  
/* Affichage d'un élément */  
void elementAfficher(ELEMENT);  
  
/* Affectation du 2eme argument dans le 1er qui est donc modifié et passé par adresse */  
void elementAffecter(ELEMENT*, ELEMENT);  
  
/* Copie du contenu du deuxième argument dans le premier, les deux arguments ont des adresses différentes  
(duplication) */  
void elementCopier(ELEMENT *, ELEMENT);  
  
/* Comparaison des arguments retourne un entier 0, < 0 ou > 0 la "différence" (e1 - e2) */  
int elementComparer(ELEMENT, ELEMENT);  
  
/* Création d'un élément */  
ELEMENT elementCreer(void);  
  
/* Libération de mémoire */  
void elementDetruire (ELEMENT);  
#endif
```

Exemple 1: stockage direct (entiers) (1)

La définition est faite dans le ELTINT.h

```
/* *****  
* Fichier : ELTINT.H  
* Contenu : Déclaration de la structure de données adoptée  
*           pour une réalisation du TDA ELEMENT.  
* ***** */  
  
#ifndef _ELTINT_H  
#define _ELTINT_H  
  
typedef int ELEMENT;  
  
#endif
```

Exemple 1: stockage direct (entiers) (2)

```
/* *****  
 * Fichier : ELTINT.C  
 * Contenu : Définition des primitives pour une réalisation par des entiers du TDA ELEMENT.  
 ***** */  
#include <stdio.h>  
#include "ELTPRIM.H"  
#define ELEMENT_VIDE 32767  
  
ELEMENT elementCreer (void) {  
    return ELEMENT_VIDE ;}  
  
void elementDetruire (ELEMENT elt) {  
    /* ne fait rien en stockage direct */  
}
```

ELEMENT est défini dans
ELTINT.H qui est reconnu
automatiquement dans
ELTINT.C

Dans le stockage direct les opérations de création et de libération sont simplifiées il n'y a ni **allocation** ni **libération** de la mémoire

Exemple 1: stockage direct (entiers) (4)

```
void elementLire(ELEMENT * elt) {  
    printf("un entier svp :") ;  
    scanf("%d",elt);  
}  
  
void elementAfficher(ELEMENT elt) {  
    printf(" %d ",elt);  
}  
  
int elementComparer(ELEMENT e1, ELEMENT e2) {  
    return (e1-e2);  
}
```

Exemple 2: stockage indirect (client) (1)

Le client est une structure :

- Nom
- Adresse
- cin

```
/* *****  
 * Fichier : ELTCLT.H  
 * Contenu : Déclaration de la structure de données adoptée  
 *           pour une réalisation du TDA ELEMENT par client.  
 * ***** */  
  
#ifndef _ELTCLT_H  
#define _ELTCLT_H  
  
typedef struct  
{  
    char nom[20];  
    char adresse[30];  
    int cin;  
} elem, *ELEMENT;  
  
#endif
```

ELEMENT ici est un pointeur sur une structure

Exemple 2: stockage indirect (client) (2)

```
/* *****  
 * Fichier : ELTCLT.C  
 * Contenu : Définition des primitives pour une réalisation par des clients du TDA ELEMENT.  
 * ***** */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include "ELTPRIM.H"  
  
#define ELEMENT_VIDE NULL  
  
ELEMENT elementCreer (void) {  
    ELEMENT L;  
    L = (ELEMENT) malloc(sizeof(elem));  
    return L;  
}  
  
void elementDetruire (ELEMENT elt) {  
    free (elt);  
}
```

Ici on utilise malloc et free car on est dans le stockage indirect (pointeurs)

Sizeof(elem) est différent de sizeof(ELEMENT)

Exemple 2: stockage indirect (client) (3)

```
void elementLire(ELEMENT* elt) {
    char x;
    printf("\nDonnez un nom svp :");
    fgets((*elt)->nom, 18, stdin);
    printf("\nDonnez une adresse svp :");
    fgets((*elt)->adresse, 18, stdin);
    printf("\nDonnez le numéro CIN :");
    scanf("%d",&((*elt)->cin));
    x=getchar();
}

void elementAfficher(ELEMENT elt) {
    printf("\nnom = %s , adresse = %s , CIN= %d ",elt->nom, elt->adresse, elt->cin);
}
```

Inclusions de bibliothèques

- La manière classique d'inclure les fichiers d'en-tête associés aux bibliothèques en C était la suivante :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

- En C++, il n'est plus nécessaire de citer l'extension **.h** pour les fichiers d'en-tête standards, mais il faut tout de même spécifier s'il s'agit d'un fichier correspondant à une librairie C, et dans ce cas on ajoute le caractère 'c' devant le nom du fichier :

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
```

Inclusions de bibliothèques

- Ancienne version en C++ (avant la normalisation)

```
#include <iostream.h> // obsolète (déconseillé)
...
cout << "message" ;
```

- Nouvelles versions (après la normalisation)

```
#include <iostream>
...
std::cout << "message" ;
```

ou bien (utilisation de l'espace de noms `std`) :

```
#include <iostream>
using namespace std;
...
cout << ...
```

Espaces de noms

Problème : un programme peut utiliser différentes bibliothèques dans lesquelles il se peut que les mêmes noms identifient des composants différents.

```
// achats.h
struct Commande { ... }; // à un fournisseur
struct Facture { ... }; // d'un fournisseur

// ventes.h
struct Commande { ... }; // d'un client
struct Facture { ... }; // à un client

// stocks.h
struct Article { ... };

// gestion.cpp
#include "achats.h"
#include "ventes.h"
#include "stock.h"
int main ()
{ Commande cmd; // Commande client ou Commande fournisseur ???
```

Comment éviter les conflits de noms ?

- En C, on était obligé de choisir des noms différents !
- En C++, le problème est résolu grâce à l'utilisation des **espaces de noms**

Espaces de noms

- Nouveau concept introduit par C++. Il s'agit de donner un nom à un regroupement de déclarations (types, fonctions, constantes ...), en procédant ainsi :

```
namespace esp_nom
{
    // déclarations
}
```

- On peut lever l'ambiguïté lorsqu'on utilise plusieurs espaces de noms comportant des identificateurs identiques ; pour cela il suffit de faire appel à l'opérateur de résolution de portée ::

```
esp_nom::ident ...
// on se réfère à l'identificateur ident de l'espace de
// noms esp_nom
```

Espaces de noms

- Pour se référer à des identificateurs définis dans un espace de noms sans recourir à l'opérateur de résolution de portée, on utilise l'instruction **using** :

```
using namespace esp_nom;
```

dorénavant, l'identificateur **ident** employé seul (sans **esp_nom::**) correspondra à celui défini dans l'espace de noms **esp_nom**.

- Remarque : Tous les identificateurs **des fichiers d'en-tête standard** sont définis dans l'espace de noms **std** ; il est donc nécessaire de recourir systématiquement à l'instruction :

```
using namespace std ;
```


Inclusions de bibliothèques

- Nouvelles versions en C++ (après la normalisation)

```
#include <iostream>
...
std::cout << "message" ;
```

ou bien (utilisation de l'espace de noms `std`) :

```
#include <iostream>
using namespace std;
...
cout << ...
```

Espaces de noms

Exemple

```
namespace Achats
{ #include "achats.h"
  // autres déclarations ...
}

namespace Ventes
{ #include "ventes.h"
  // autres déclarations ...
}

namespace Stocks
{ #include "stocks.h"
  // autres déclarations ...
}

int main ()
{
  /* 1) désignation explicite de
    composants appartenant à des
    espaces différents */
  Achats::Commande cde_fourn;
  Ventes::Commande cde_client;

  /* 2) importation locale d'un
    identifiant de composant : */
  Ventes::Facture;
  Facture fact_client;

  /* 3) mise à disposition de tous les
    identifiants d'un espace de noms */
  using namespace Stocks;
  Article prod;
```

Espaces de noms

Exemple

```
namespace versionC
{ void permut(int* a, int* b)
  { int tmp = *a;
    *a = *b;
    *b = tmp;
  }
}

namespace versionCPP
{ void permut(int& a, int& b)
  { int tmp = a;
    a = b;
    b = tmp;
  }
}

int main ()
{
  int i=5, j=2;
  versionC::permut(&i,&j);
  cout << "i=" << i << " j=" << j;
  cout << endl;

  versionCPP::permut(i, j);
  cout << "i=" << i << " j=" << j;
  // ou bien
  // using namespace versionCPP;
  // permut(i, j);
}
```

Espaces de noms

Exemple

```
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main ()
{
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}

// Résultat d'affichage :
// 5
// 10
// 3.1416
// 2.7183
```