La Programmation Orientée Objet C++

Fatma CHAKER KHARRAT

M1 MMD
Tunis Dauphine
Année universitaire 2014/2015

Séance 1: Notions de base

- Petit historique
- Bibliographie
- Présentation du C++
- La Structure d'un programme C++
- Le préprocesseur
- Les types élémentaires en C++
- La déclaration des variables
- Les opérateurs
- Les structures de contrôle
- Les fonctions

Petit historique

- **3ème langage le plus utilisé au monde (classements TIOBE de Août 2013 *http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html) et LangPop de Avril 2013 http://langpop.com/)***
- **JVM (HotSpot) et une partie du noyau de Google Chrome écrits en C++**
- Première version développée par Bjarne Stroustrup de Bell Labs AT&T en 1980
- Appelé à l'origine « Langage C avec classes »
- Devenu une norme ANSI/ISO C++ en juillet 1998 (C++98 - ISO/IEC 14882) – mise à jour en 2003 (C++03)

ANSI : American National Standard Institute
ISO : International Standard Organization

Nouvelle norme C++: C++11

- C++11 (C++0x) approuvée par l'ISO en 12/08/2011 et disponible depuis sept. 2011 (norme ISO/CEI 14882:2011)
- Quelques sites explicatifs des nouveautés de la norme 2011 :
  - http://www.siteduzero.com/tutoriel-3-497647-introduction-a-c-2011-c-0x.html
  - http://en.wikipedia.org/wiki/C%2B%2B11
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
  - http://www.stroustrup.com/C++11FAQ.html
  - http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style
  - C++11 improvements over C++03 - http://www.cplusplus.com/articles/EzywvCM9/

Nouvelle norme C++: C++14

- C++14 : révision mineure de C++11
  - http://electronicdesign.com/dev-tools/bjarne-stroustrup-talks-about-c14
  - https://parasol.tamu.edu/people/bs/622-GP/C++14TAMU.pdf
  - http://www1.cs.columbia.edu/~aho/cs4115/lectures/14-01-29_Stroustrup.pdf
- Nouvelle mise à jour annoncée pour 2017
- Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* - O'Reilly Media; 1 edition (December 5, 2014)ISBN-13: 978-1491903995

Bibliographie

- ***Le Langage C++ (The C++ Programming Language), de Bjarne Stroustrup, Addisson-Wesley – 4ème édition, mai 2013***
- Programmation: Principes et pratique avec C++, , de Bjarne Stroustrup et al., Pearson Education, déc. 2012
- ***How to program – C and introducing C++ and Java, de H.M. Deitel et P.J. Deitel, Prentice Hall, 2001 – dernière édition C++ How To Program de février 2005***
- ***Programmer en langage C++, 8ème Édition de Claude Delannoy, Eyrolles, 2011***
- ***Exercices en langage C++, de Claude Delannoy, Eyrolles, 2007***

Documents en ligne

- *Petit manuel de survie pour C++ de François Laroussinie, 2004-2005,* ***http://www.lsv.ens-cachan.fr/~fl/Cours/docCpp.pdf***
- *Introduction à C++ de Etienne Alard, revu par Christian Bac, Philippe Lalevée et Chantal Taconet, 2000* ***http://www-inf.int-evry.fr/COURS/C++/CoursEA/***
- ***Thinking in C++ de Bruce Eckel, 2003 http://w2.syronex.com/jmr/eckel/***
- **http://www.stroustrup.com/C++.html**
- **http://channel9.msdn.com/Events/GoingNative/2013/Opening-Keynote-Bjarne-Stroustrup**
- **Livres gratuits en ligne : http://itebooks.info/book/1256/**
- **Aide en ligne : http://www.cplusplus.com/doc/tutorial/**

Présentation du C++

Le C++ est un langage **multiparadigme**. Il supporte essentiellement les paradigmes suivants :

- **Programmation procédurale** : il reprend essentiellement les concepts du langage C, notamment la notion de fonction (une procédure étant une fonction avec un retour de type 'void') ;
- **Programmation structurée** : il reprend la notion struct du langage C. Cette notion est considérée en C++ aussi comme des classes dont l'accès par défaut est public ;
- **Programmation orientée-objet** : il implémente la notion de **classe**, **d'encapsulation**, **d'héritage** (simple ou multiple), **d'abstraction** grâce aux classes de base abstraites pures, de **polymorphisme dynamique** (ou au runtime) grâce aux fonctions membres virtuelles ;
- **Programmation générique** : il implémente les fonctions et classes génériques.

C++ versus C

Par rapport au **langage C**, le **C++** apporte :

- les **concepts orientés objet** (encapsulation, héritage) ;
- les **références** ;
- la **vérification stricte des types** ;
- les **valeurs par défaut** des paramètres de fonctions ;
- la **surcharge de fonctions**;
- la **surcharge des opérateurs** (pour utiliser les opérateurs avec les objets) ;
- les **templates** de classes et de fonctions ;
- les **constantes typées** ;
- la possiblité de **déclaration de variables** entre deux instructions d'un même bloc.

Programmation C++ : Compilation

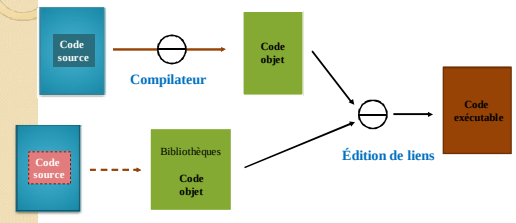
La compilation : Série d'étapes de **transformation** du **code source** en du **code machine** exécutable sur un processeur cible.

Le **langage C++** fait partie des **langages compilés** : le fichier exécutable est produit à partir de **fichiers sources** par **un compilateur**.

Programmation C++ : Phases de Compilation

- **Préprocessing** : Le **code source original** est **transformé** en **code source brut**. Les commentaires sont enlevés et les directives de compilation commençant par # sont d'abord traités pour obtenir le **code source brut** ;
- **Compilation en fichier objet** : les **fichiers de code source brut** sont **transformés** en **un fichier dit objet**, c'est-à-dire un fichier contenant du code machine ainsi que toutes les informations nécessaires pour l'étape suivante. Généralement, ces fichiers portent l'extension .obj ou .o ;
- **Edition de liens** : dans cette phase, l'éditeur de liens (linker) s'occupe d'assembler les fichiers objet en une entité exécutable. L'entité exécutable est généralement soit un exécutable, soit une bibliothèque dynamique (DLLs sous windows et toutes les variantes, tels que objet COM, OCX, etc, et les .so sous linux).

La compilation comment ça marche?



La compilation : Erreurs générées

- Erreurs de compilation
- Erreurs de syntaxe, déclaration manquante, parenthèse manquante,...
- Erreur de liens
- Appel à des fonctions dont les bibliothèques sont manquantes
- Erreurs d'exécution
- *Segmentation fault, overflow, division par zéro*
- Erreurs logiques

La compilation : Exemple de compilateurs

- **Linux**
  - Le compilateur C le plus utilisé est GCC
  - Son équivalent C++ est G++
- **Windows**
  - GCC/G++ existent avec Cygwin et MinGW
  - Différents IDE existent et fournissent leurs propres compilateurs
    - Microsoft Visual Studio avec CL
    - Borland C++ Builder / Turbo C++ / Borland Developper Studio avec BCC32
    - Code Blocks / Dev-C++ avec MinGW

C++ : Les fichiers sources

- Les fichiers .cpp (parfois .c ou .cc) contiennent la **définition** (l'implémentation) des différentes fonctions et méthodes définies dans les fichiers d'en-tête. La compilation des fichiers .cpp produit dans un premier temps des fichiers objets (extension .obj ou .o en général).
- Ces fichiers .cpp utilisent les fichiers d'en-tête. Ils les appellent en utilisant la syntaxe **#include "nomdefichier »**.

C++ : Les fichiers entêtes

Les fichiers "**entêtes**" ("**headers**" en anglais), traditionnellement d'extension **.h** ou **.hpp** contiennent généralement les **prototypes** de différentes **fonctions**, **structures** et **classes**. Ces fichiers sont inclus dans les fichiers sources à l'aide de la directive **#include « nom_header.h"** .

Les prototypes proviennent :

- **des bibliothèques standards du C++**
- **de bibliothèques non standards fournis par l'éditeur du compilateur ou de l'environnement de développement ;**
- **de bibliothèques non standards (gratuites ou payantes) que le programmeur s'est procuré :** citons par exemple la bibliothèque permettant d'accéder à une base de données mySQL.

C++ : Un programme minimal

```
#include <iostream>

int main()
{
    std :: cout << "BONJOUR" << std:: endl;
    return 0;
}
```

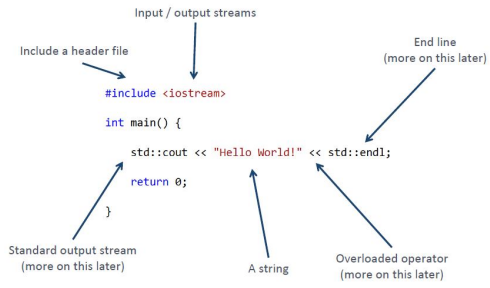
Inclusion de la bibliothèque standard (ANSI) du C++ pour les E/S

Déclaration de fonctions

Affichage à l'écran BONJOUR

Fin de la fonction main. Tout s'est bien passé

C++ : Un programme minimal



C++ : Un peu de syntaxe

- Toute déclaration/instruction se termine par un **point-virgule**.
- Instruction ≠ INSTRUCTION
- Les commentaires:

```
\* Ceci est un commentaire sur plusieurs lignes
...
Fin du commentaire*/

//ceci est un commentaire sur une ligne
```

- Saisie et affichage des données (communication avec les E/S):
  - Saisir une variable au clavier: **cin**

syntaxe : **cin>>exp$_1$>>exp$_2$>>...>> exp$_n$;**

Bibliothèque iostream

  - Afficher une variable ou une phrase à l'écran: **cout**

syntaxe : **cout<<exp$_1$<<exp$_2$<<...<<exp$_n$;**

C++ : Un peu de syntaxe

Plus besoin d'utiliser std::

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

int main()
{
   cout << "BONJOUR" << endl;

   return 0;
}
```

```
#include <iostream>
int main()
{
  std:: cout << "BONJOUR" <<
std::endl;

  return 0;
}
```


C++ : Le préprocesseur

Avant de compiler le programme, il est possible d'effectuer certaines modifications sur le code source. Le programme effectuant ces modifications s'appelle le **préprocesseur**.

Les commandes destinées au préprocesseur commencent toutes par **#** en début de ligne.

Le préprocesseur : Inclusion de fichiers

Syntaxe :

#include "*nom_du_fichier*" — Le fichier se trouve dans le même dossier que le fichier source

Ex: #include " MaClasse.h "

#include <*nom_du_fichier*> — Le fichier se situe dans un endroit différent (ce fichier pouvant être fournit par le compilateur ou une librairie externe par exemple)

Ex: #include<iostream>

Le préprocesseur : #define, #undef

❑ La directive **#define** permet de remplacer toutes les occurrences d'un certain mot par un autre.

Exemple : #define N 1143
#define PI 3.141162654

→ **const** int N=1143;
const PI=3.141162654;

❑ La directive **#undef** permet d'arrêter une définition.

Exemple : #undef N

Le préprocesseur : Les macros

Les macros sont des **#define** particuliers qui contiennent des paramètres.

Exemple :

```
#include <iostream>
using namespace std;
#define Coucou() cout<<« Hello »;
#define CARRE(a)  cout<<a*a;
int main()

 { Coucou()
CARRE(5)
return 0;
}
```

Le préprocesseur : #if, #elif et #endif

```
#if condition //Cette condition ne peut utiliser que des constantes
              //définies par une directive #define
/* Code source à compiler si la condition est vraie */
#elseif condition2
/* Sinon si la condition 2 est vraie compiler ce code source */
#endif
```


Le préprocesseur : #if, #elif et #endif

```
#define N_ORANGES 5
// - de 3 variétés de pommes
#define N_POMMES 3
#define N_TOTAL_FRUITS N_ORANGES+N_POMMES double
prix_fruits[N_TOTAL_FRUITS];
#if N_TOTAL_FRUITS > 7
// ... plus de 7 variétés de fruits
#elseif
// ... moins de 7 ou égale à 7 variétés de fruits
#endif
```


Le préprocesseur : #ifdef, #ifndef

Il est possible d'utiliser :

- **#ifdef** pour dire « Si la constante est définie ».
- **#ifndef**, lui, sert à dire « Si la constante n'est pas définie ».

```
#define WINDOWS
#ifdef WINDOWS
/* Code source pour Windows */
#endif

#ifdef LINUX
 /* Code source pour Linux */
#endif

#ifdef MAC
 /* Code source pour Mac */
 #endif
```

Adapter un programme multi-plates-formes à l'OS.

Le préprocesseur : #ifndef

#ifndef pour éviter les inclusions infinies

#ifndef est très utilisé dans les **.h** pour éviter les « inclusions infinies ».

```
#ifndef NOMDUFICHIER_H
// Si la constante n'a pas été définie le fichier n'a jamais été inclus

#define NOMDUFICHIER_H
// On définit la constante pour que la prochaine fois le fichier ne soit plus inclus

        /* Contenu de votre fichier .h (autres include, prototypes,
        define...) */
#endif
```


Le préprocesseur : A retenir

- Le préprocesseur est un programme qui **analyse** votre code source et y effectue des modifications **avant la compilation**.
- L'instruction du préprocesseur **#include** insère le contenu d'un autre fichier.
- L'instruction **#define** définit une constante de préprocesseur. Elle permet de remplacer un mot-clé par une valeur dans le code source.
- Les **macros** sont des **morceaux de code** tout prêts définis à l'aide d'un **#define**. Ils peuvent accepter des paramètres.
- Il est possible d'écrire des conditions en langage préprocesseur pour choisir ce qui sera compilé. On utilise notamment les mots-clés **#if, #elif et #endif**.
- Pour **éviter** qu'un fichier .h ne soit **inclus un nombre infini de fois**, on le protège à l'aide d'une **combinaison de constantes de préprocesseur et de conditions.** Tous vos futurs fichiers .h devront être protégés de cette manière

Les types élémentaires en C++

- Les **entiers relatifs** (1,2,-3 etc...) : **int** (2 octet). (on peut leur ajouter les attributs : short, long, signed, unsigned)
- Les **réels ou flottants** (0.12, PI, ...) : **float** (4 octet).
- Les **réels double précision** : **double** (8 octets). (on peut lui ajouter l'attribut : long)
- Les **caractères** ('a', 'b', 'c', '#', '3'...) : **char** (1octet) = code ASCII (0<<255) du caractères. (on peut lui ajouter les attributs signed ou unsigned)
- Le type **logique ou booléen** (true, false) : **bool**

Les variables : Déclaration

Nom_type nom_objet;

Ex : int a;

➢ Une variable peut être **initialisée lors de sa déclaration**, **deux notations sont possibles :**

int p=34;
double x=12.34;

ou

int p(34);
double x(12.34);

➢ Une variable d'un type élémentaire **non initialisée n'a pas de valeur définie**, i.e. peut contenir n'importe quoi.

Les variables : Déclaration

C++ permet de déclarer des variables n'importe où et de les initialiser.

Règles:

- toujours déclarer ses variables au dernier moment.
- déclarer une variable par ligne.

Les variables : Les constantes symboliques

Il est possible de définir des constantes symboliques avec le mot clé **const** :

Syntaxe :

const type nom = val;

Exemple :

```
const int Taille = 100 ;    //remplace #define Taille 100;
```

Rque : Il ne sera **pas possible de modifier** Taille dans le reste du programme (erreur à la compilation).

Les Tableaux : Déclaration

Type nom[taille];

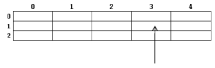
Exemple :

```
int Tab[32];
float salaire[32];
```

- Le type est le même pour chaque élément du tableau
- La taille du tableau doit être exprimé avec un chiffre pour pouvoir compiler

```
#define MAX_ELEM 32    // ou encore const int MAX_ELEM=32;
float salaire[MAX_ELEM];
```

```
int tab_a_2dim[3][5];
```



tab_a_2dim[1][3]

Déclaration des variables

- Déclaration **locale** ou **globale**:
  - **Variables globales :**
    - définies **hors de toutes fonctions**
    - **Vues** par **toutes les fonctions du fichier source**
    - **Persistante** pendant **toute la durée de vie** du programme
  - **Variables locales:**
    - **Déclarées** à l'intérieur d'une fonction
    - **Détruites en sortie** de fonction

```
#include <iostream.h>

const float PI = 3.14159;

int main ()
{
  float  rayon = 5.458;
  float  perimetre;

  // calcul périmètre
  perimetre = PI * 2* rayon;

  cout << "perimetre p = " << p << endl;

  return 0;
}
```

déclaration de variables globales

déclaration de variables locales

Les opérateurs

- Affectation: =
- Opérateurs arithmétiques
  - Addition: +
  - Soustraction: -
  - Multiplication: *
  - Division: / (division entière dans le cas de int)
  - Modulo: % (reste de la division entière)
- Opérateurs arithmétiques unaires (un seul opérande)
  - Moins unaire: -
  - Incrémentation: ++
  - Decrémentation: --
- Opérateurs logiques
  - Négation: !
  - ET logique: &&
  - OU logique: ||
- Opérateurs de relation (ils rendent 1 si la condition est vrai et 0 sinon)
  - < , > , <= , >= , == , !=

Les opérations mathématiques de base

```
int i = 100 + 50;
int j = 100 - 50;
int n = 100 * 2;
int m = 100 / 2; // division entière
int k= 100 % 2; // modulo – reste de la division entière

i = i+1;
i = i-1;

j++; // équivalent à j = j+1;
j--; // équivalent à j = j-1;

n += m; // équivalent à n = n+m;
m -= 5; // équivalent à m = m-5;
j /= i; // équivalent à j = j/i;
j *= i+1; // équivalent à j = j*(i+1);

int a, b=3, c, d=3;
a=++b; // équivalent à b++; puis a=b; => a=b=4
c=d++; // équivalent à c=d; puis d++; => c=3 et d=4
```


Les opérateurs de comparaison

```
int i,j;
…
if(i==j) // évalué à vrai (true ou !=0) si i égal j
{
  … // instructions exécutées si la condition est vraie
}

if(i!=j) // évalué à vrai (true ou !=0) si i est différent de j

if(i>j) // ou (i<j) ou (i<=j) ou (i>=j)


if(i) // toujours évalué à faux si i==0 et vrai si i!=0
if(false) // toujours évalué à faux
if(true) // toujours évalué à vrai
```

Ne pas confondre = (affectation) et == (test d'égalité)

```
if (i=1) // toujours vrai et i vaut 1 après
```


Les opérateurs : L'affectation

```
#include <iostream>
using namespace std;
int main()
{
 int a, b, s;
cout << "Tapez la valeur de a : ";
cin >> a;
cout << "Tapez la valeur de b : ";
cin >> b;
s = a + b;
// affecter le résultat de l'addition à la
//variable s
 cout << "La somme a+b vaut : " << s <<
endl;
return 0;
}
```

Exécution
Tapez la valeur de a : **45**
Tapez la valeur de b : **67**
La somme a+b vaut 112

Les opérateurs : L'affectation

Affectation en série

Le résultat d'une même expression assigné à plusieurs variables, en **une seule instruction** :

Syntaxe :

identificateur2=identificateur1=expression

Exemple :

a = b = 38 + t; **// qui est équivalent à : a = (b = 38 + t);**

Le résultat de 38+t est calculé ;
Il est affecté à la variable b ;
Il est retourné par l'opérateur d'affectation ;
Cette valeur retournée est affectée à la variable a.

Les opérateurs : L'affectation

Affectation à la déclaration

L'affectation à la déclaration d'une variable est appelée "déclaration avec initialisation« .

Exemple :

int **a** = 57;

Ne pas confondre = (affectation) et == (test d'égalité)

Les opérateurs arithmétiques

***, +, -, / (division entière et réelle), % (modulo ne peut porter que sur des entiers).**

Remarque : Lors de l'´evaluation d'une (sous)expression mélangeant plusieurs types, on utilise la règle des **conversions implicites**.

Int ⇨ **long** ⇨ **float** ⇨ **double** ⇨ **long double**

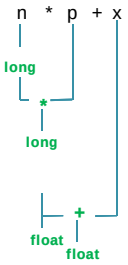
Exemple :

```
//=================
int i=3,j=2,m;
double r=3.4;
m = (i/j)*r;
//=================
```

m s'appelle une **lvalue**. Les règles de conversions implicites **ne sont pas appliquées** et le résultat doit être **converti dans le type de la lvalue.**

Les conversions implicites en C++

Exemple :
```
int n;
long p;
float x;
n*p+x
```



Les conversions systémiques ou promotions numériques en C++

Si l'un des opérandes est de type short, char ou bool, C++ prévoit de le convertir en int.

Exemple 1:
```
short  p1,p2,p3;
float  x;
p1*p2+p3*x;
```

Exemple 2 :
```
bool   OK=true;
.....
cout<<OK+2     //
.....
OK=false;
cout<<OK+2     //
```


Les opérateurs logiques

Négation: !
ET logique: &&
OU logique: ||

Exemple :
```
bool b;
int u = 18;
b = !( (u>20 || (u<0)));  // b vaut true.
bool b = ! false ;        //  b vaut true
b = ! true ;              // b vaut false
```


Priorité des opérateurs

- x[y] x++ x--
- ++x --x !x -x
- x*y x/y x%y
- x+y x-y
- x >> y x << y
- x < y x > y x >= y x <= y
- x == y x != y
- x && y
- x || y
- x = y x op= y
- x? y : z

Dans le cas où les opérateurs ont la **même priorité**, l'expression est évaluée de **gauche à droite**.

Exercices : Concepts de base

Exercice :
Ecrire un programme qui demande à l'utilisateur de taper 5 entiers et qui affiche leur moyenne. Le programme ne devra utiliser que 2 variables.

Structures de contrôle : L'instruction If

```
If (expression)
        instruction1;
else
        instruction2;
```

```
If (expression)     instruction1;
```

```
#include<iostream>
using namespace std;
int main()
{
float a;
cout << "Entrer un réel :";
cin >> a;

if(a > 0) cout << a << " est positif\n";
else
    if(a == 0) cout << a << " est nul\n";
    else cout << a << " est  négatif\n";
}

/* Mettre des {} pour les blocs d'instructions des if/else pour éviter les ambiguïtés et lorsqu'il y a plusieurs instructions */
```

Structures de contrôle : L'instruction If

(exp) ? exp1 : exp2;

```
x = 10;
y = x > 9 ? 100 : 200;
```

Equivalent à :

```
if(x>9) y=100;
else
y=200;
```


Structures de contrôle : Switch

Syntaxe :

```
switch(expressions)
{
   case const1: instruction1;
                break;
   case const2: instruction2;
                break;
   -----
   case constn: instructionn;
                break;
  [default] : instructions;
}
```

```
int main()
{ int ch;
double x=5.0, y=10.0;
cout << " 1. Afficher la valeur de x\n";
cout << " 2. Afficher la valeur de y\n";
cout << " 3. Afficher la valeur de x*y\n";
cin >> ch;
switch(ch)
{   case 1: cout << x << "\n";
            break; // pour sortir du switch
    // et ne pas exécuter les commandes suivantes
    case 2: cout << y << "\n";
            break;
    case 3: cout << x*y << "\n";
            break;
    default: cout << « Option non reconnue \n";
} \\ Fin du switch
} \\ Fin du main
```

Structures de contrôle

Boucle While

Syntaxe :

```
While (condition)
   expression;
```

Boucle Do.. while

Syntaxe :

```
Do
   expression;
While (condition);
```

Boucle FOR

Syntaxe :

```
For(exp1;exp2;exp3)
   expression4;
```


Structures de contrôle : FOR

Remarques:

❶ **Chacune des 3 expressions est facultative**

```
#include<iostream>
using namesapce std;
int main()
{
int i;
for(i=1;i<=5;i++)
cout<<« Bonjour » <<i<<
« fois »<<endl;
}
```

≅

```
#include<iostream>
using namesapce std;
int main()
{ int i=1;
for(;i<=5;i++)
cout<<« Bonjour » <<i<<
« fois »<<endl;
}
```

```
#include<iostream>
using namesapce std;
int main()
{int i=1;
for(;i<=5;)
cout<<« Bonjour » <<i<<
« fois »<<endl;
i++;
}
```


Structures de contrôle : FOR

❷ **exp1, exp2 et exp3 peut regrouper plusieurs actions.**

```
int i,j;
float k;
....
For(i=1,j=0,k=1.5;...;...)
```

❸ **exp1 peut contenir à la fois une déclaration et une initialisation**

Exemple:

```
For(int i=1;...;....)
```

```
For(int i=1,j=0,float k=1.5;...;...)
```

FAUX
1 seule initialisation

Les contrôles d'itérations : break

L'instruction **break** sert à **mettre fin ou interrompre** une **boucle** (for, while et do), ou un **switch**.

```
#include<iostream>
using namespace std;
int main()
{
for(int i=1;i<=10;i++)
  {       cout<<« Début tour »<<i<<endl;
          if(i==3) break;
          cout<<« Fin tour »<<i<<endl;
  }
cout<<« Fin boucle »;
}
```

Exécution ?

Les contrôles d'itérations : continue

Continue permet de passer à l'itération suivante de la boucle la plus imbriquée.

```
#include<iostream>
using namespace std;
int main()
{
for(int i=1;i<=5;i++)
  {        cout<<« Début tour »<<i<<endl;
           if(i<4) continue;
           cout<<« Bonjour »<<endl;
 }
cout<<« Fin boucle »;
}
```

Exécution ?

Les contrôles d'itérations : goto

L'instruction **goto** permet le branchement en un emplacement quelconque du programme repéré par une **étiquette.**

Syntaxe :

goto étiquette

étiquette : instruction;

```
#include<iostream>
using namespace std;
int main()
{
for(int i=1;i<=10;i++)
  {        cout<<« Début tour »<<i<<endl;
           cout<<« Bonjour »;
           if(i==3) goto sortie;
           cout<<« Fin tour »<<i<<endl;
 }
sortie : cout<<« Fin boucle »;
}
```


Exercices : Structures de contrôle

Exercice 1:
Ecrire un programme qui demande à l'utilisateur de taper 10 entiers et qui affiche le plus petit de ces entiers.

Solutions Exercices : Exercice 1

```
#include<iostream>
using namespace std;
int main()
{
int Nbre, Min;
cout <<"Donner le 1er nombre ";
cin>>Nbre;
Min=Nbre;
for (int i=2;i<10;i++)
{ cout<<"\n Donner le "<<i<<" eme nombre ";
cin>>Nbre;
if (Nbre<=Min)
Min=Nbre;
}
cout<<"Le minimum de cette liste de valeur est :"<<Min<<endl;
system("PAUSE");
return 0;
}
```


Les fonctions en C++

Définition d'une fonction :

```
type nom(liste des paramètres)
{
        Corps de la fonction
}
```

• **type** est le type du résultat renvoyé par la fonction (void s'il n'y a pas de valeur de retour).
• **Liste des paramètres** (**paramètres formels**) de la forme :

$type_1\ p_1, \ldots, type_n p_n$

• **Corps de la fonction** décrit les instructions à effectuer lors de l'appel de cette fonction.

NB : Lorsqu'une fonction renvoie un résultat, il doit y avoir (au moins) une instruction "**return expr ;**"

Les fonctions en C++

Appel d'une fonction :

```
nom(liste des arguments)
```

Liste des arguments (**paramètres réels**) est de la forme :

$expr_1, expr_2, \ldots expr_n$

où chaque $expr_i$ est compatible avec le type $type_i$ du paramètre formel p_i.

Exemple :

```
//=========================
// Définition de la fonction max
int max(int a,int b)
{ int res=b;
if (a>b) res = a;
return res;
}
//=========================
```

```
//=========================
// Appel de la fonction max
int k=34, t=5, m;
m = max(k,2*t+5);
//=========================
```

Les fonctions en C++

Un **appel de fonction** doit **toujours être précédé** par la **définition OU** une **déclaration** de la fonction.

Une déclaration consiste à donner : son type, son nom et le type des paramètres (le nom des paramètres n'est pas nécessaire) suivi d'un point virgule :

Déclaration d'une fonction (prototype):

```
type nom(types des paramètres);
```

Exemple :

```
int max(int ,int ) ; // prototype
```


Les fonctions en C++

Sans prototype :

```
//=========================
// Définition de la fonction max
int max(int a,int b)
{ int res=b;
if (a>b) res = a;
return res;
}
//=========================
//Programme principal
//Appel de la fonction max
int main( )
{ …
int k=34, t=5, m;
m = max(k,2*t+5);
…
}
```

Avec prototype :

```
//=========================
// déclaration du prototype
int max(int, int);
//---------------------
int main( )
{ int k=34, t=5, m;
m = max(k,2*t+5);
//=========================
//=========================
// Définition de la fonction max
int max(int a,int b)
{ int res=b;
if (a>b) res = a;
return res;
}
//=========================
```


Les fonctions en C++ : Le return

- L'instruction return : fournir une **valeur de retour** et **mettre fin à l'exécution** de la fonction.
- Fonction ne fournit **aucune valeur de retour** : l'entête et la déclaration comportent le mot clé « **void** ».

  Ex: **void** sans_valeur(int n) // en-tête
  void sans_valeur(int); // déclaration
- Lorsqu'une fonction ne reçoit **aucun argument** : liste vide (Ex : fonction renvoyant un nbre aléatoire)

  Ex: **float** tirage( ) **// en-tête de la fonction**
  float tirage( ); **// sa déclaration**
- Fonction sans valeur de retour et sans arguments

  Ex : **void** message ( ) **// en-tête**
  void message ( ); **// sa décalartion**

NB : en C on utilise le mot clé **void** à la place de la liste vide ( ).

Les fonctions : Arguments par défaut

- En C++, dans la déclaration d'une fonction (**prototype**) il est possible de prévoir des **valeurs par défaut pour un ou plusieurs arguments** (**obligatoirement les derniers de la liste**)

Exemple : float fct(char, int =10, float = 1.2);

Ces valeurs par défaut seront utilisées lorsqu'on appelle la fonction :

Exemple : fct('a') //est équivalent à fct('a',10,1.2)
fct('a',12) // est équivalent à fct('a',12,1.2)

Exemple : Arguments par défaut

Calculer le nombre de secondes en fonction d'un nombre d'heures, de minutes et de secondes transmis !

```
#include <iostream>
using namespace std;
// Prototype avec les valeurs par défaut
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);
int main()
{
cout << nombreDeSecondes(1, 10, 25) << endl;     // résultat = 4225=3600+600+25
cout << nombreDeSecondes(1, 10) << endl;         // résultat = 4200
cout << nombreDeSecondes(1)<< endl;              // résultat = 3600
cout << nombreDeSecondes()<< endl;               // résultat = Erreur
return 0;
}
// Définition de la fonction, SANS les valeurs par défaut
int nombreDeSecondes(int heures, int minutes, int secondes)
{ int total = 0;
total = heures * 60 * 60;
total += minutes * 60;
total += secondes;
return total;
}
```

A retenir

✓ *les valeurs par défaut sont spécifiées **uniquement dans le prototype**, **pas dans la définition de la fonction !***
Si votre code est découpé en plusieurs fichiers, alors il ne faut spécifier les valeurs par défaut que dans **le fichier d'en-tête .h.**

✓ *Les paramètres facultatifs doivent **obligatoirement** se trouver à **la fin (à droite).***

FAQ

Et si je veux envoyer à la fonction juste les heures et les secondes, mais pas les minutes ?

Tel quel **impossible**. Le compilateur analyse les paramètres **de gauche à droite**

Est-ce que je peux rendre seulement les heures facultatives, et rendre les minutes et secondes obligatoires ?

Si le prototype est défini dans le même ordre que tout à l'heure : **NON**.
Les paramètres facultatifs doivent obligatoirement se trouver à la fin.

Code : C++

```
int nombreDeSecondes(int heures = 0, int minutes, int secondes);
//Erreur, les paramètres par défaut doivent être à droite
```

La solution, pour régler ce problème, consiste à placer le paramètre heures à la fin :

Code : C++

```
int nombreDeSecondes(int secondes, int minutes, int heures = 0);
//OK
```


FAQ

Est-ce que je peux rendre tous mes paramètres facultatifs ?

Oui, cela ne pose pas de problème :

Code : C++

```
int nombreDeSecondes(int heures = 0, int minutes = 0, int secondes =
0);
```

Dans ce cas, l'appel de la fonction pourra s'écrire comme ceci :

```
cout << nombreDeSecondes() << endl;
```


Les fonctions : passage des paramètres

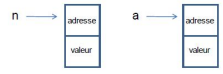
Transmission par valeur

```
#include<iostream>

void echange (int a, int b)
{ int c;
  cout << "debut echange : " << a << " " << b << endl;
  c = a;
  a = b;
  b = c;
  cout << "fin echange : " << a << " " << b << endl;
}

main ()
{ int n = 10, p = 20;
  cout << "avant appel : " << n << " " << p << endl;
  echange (n, p);
  cout << "apres appel : " << n << " " << p << endl;
}
```

avant appel : 10 20
début échange : 10 20
fin échange : 20 10
après échange : 10 20



les valeurs de n et p sont dupliquées
on change les valeurs des copies

Les fonctions : passage des paramètres

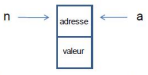
Transmission par référence

```
#include<iostream>

void echange (int& a, int& b)
{ int c;
  cout << "debut echange : " << a << " " << b << endl;
  c = a;
  a = b;
  b = c;
  cout << "fin echange : " << a << " " << b << endl;
}

main ()
{ int n = 10, p = 20;
  cout << "avant appel : " << n << " " << p << endl;
  echange (n, p);
  cout << "apres appel : " << n << " " << p << endl;
}
```

avant appel : 10 20
début échange : 10 20
fin échange : 20 10
après échange : 20 10



les valeurs de n et p ne sont pas dupliquées
on change les valeurs des originaux

Les références constantes

- le passage par référence est efficace — pas de recopie
- idée : utiliser des références tout en s'assurant qu'elles ne seront pas modifiées
- pour cela on utilise le qualifieur `const`

Exemple :

```
1 void foo(const Type& arg) {
2   ... // arg ne peut pas etre modifie
3 }
```

```
void f1(string texte); //Implique une copie coûteuse de 'texte'
{…}
void f1(string& texte);  //Implique que la fonction peut modifier texte
{…}
void f1(string const& texte); //Pas de copie et pas de modification possible
{…}
```


Les fonctions : passage des paramètres

Les pointeurs permettent à des fonctions d'accéder aux données elles même et non à des copies.

Transmission par adresse

```
#include<iostream>

void echange (int* a, int* b)
{ int c;
  cout << "debut echange : " << *a << " " << *b << endl;
  c = *a;
  *a = *b;
  *b = c;
  cout << "fin echange : " << *a << " " << *b << endl;
}

main ()
{ int n = 10, p = 20;
  cout << "avant appel : " << n << " " << p << endl;
  echange (&n, &p);
  cout << "apres appel : " << n << " " << p << endl;
}
```

avant appel : 10 20
début échange : 10 20
fin échange : 20 10
après échange : 20 10

les valeurs de n et p ne sont pas dupliquées
on change les valeurs des originaux

Les fonctions : Surcharge de fonction

- permet de réaliser le "polymorphisme ad hoc"
- **Mécanisme** :
  - ★ un même nom désigne différentes fonctions dans une même portée
  - ★ le compilateur choisit la fonction à appliquer en fonction du type des arguments
- simplifie la tâche du programmeur qui n'a plus à mémoriser et à taper des noms barbares

Les fonctions : Surcharge de fonction

6. Surdéfinition de fonctions

une fonction avec un même nom possède plusieurs significations

```
#include <iostream>

void sosie (int a)
{ cout << "sosie numero 1 : a = " << endl;
}


void sosie (double a)
{ cout << "sosie numero 2 : a = " << endl;
}


main ()
{ int n = 5;
  double x = 2.5;

  sosie (n);
  sosie (x);
}
```

sosie numero 1 : a = 5
sosie numero 2 : a = 2.5

Que se passe t il si x est de type float, long double, …?
⟹ conversion du type par le compilateur

Pile d'appels des fonctions

- pendant l'exécution les fonctions sont mémorisés dans une pile
- quand une fonction est appelée, cet appel est mémorisé sur cette pile
- quand une fonction termine (retourne), elle est dépilée
- la fonction en haut de pile est la fonction courante

Exemple

```
1  void f1() {
2    f2();
3    f3();
4  }
5
6  void f2() {
7  }
```

```
8  void f3() {
9  }
10
11 int main() {
12   f1();
13   return 0;
14 }
```