Programmation orientée objet (C++)

Adil JEGHAL
adil.jeghal@usmba.ac.ma

Génie du développement numérique et Cybersécurité

C++ versus C

Le C++ est un langage structuré, typé et modulaire, autorisant la programmation orientée objet

Principal avantage : compatibilité C/C++

- même syntaxe de base
- code C "propre" directement compilable en C++
- facilité d'intégration de fichiers C++ et C dans un même programme

Principal inconvénient : quelque incompatibilité C/C++

Ressemblances

- syntaxe en partie similaire
- fonctionnalités objet de même nature

Différences

- héritage multiple
- redéfinition des opérateurs

Exemple ‘hello c++ word’

```
// Helloword.Cpp : " salut le monde " en C++
//
#include <iostream>  //on u
using namespace std; //le nom d'espace contenant cout et cin

/* Ce commentaire multi-ligne précè
La fonction 'main' , qui e
Le programme principal
*/


int main( )
{

      cout<<"hello C++ world!\n";


      Return 0;


}
```

commentaire

Directive du processeur inclure fichier .h

la fonction main retourne un entier

Une seule fonction main par programme C++

<<opérateur d ’insertion de stream

n se termine par ;

Une instr

std::cout

Espaces de noms

Lorsque plusieurs bibliothèques sont utilisées, deux fonctions différentes peuvent avoir le même nom sans faire la même chose. Pour éviter ce problème, le C++ définit les **espaces de noms** :

```
namespace MaBiblio{
fonc1() ;
fonc2() ;
...}
```

Cela ressemble à une classe mais ce n'est pas le cas

- Il n'y a pas d'objets
- les variables définies dans un espace de noms sont des variables globales, pas des attributs

Espaces de noms

Un espace de nom à un comportement similaire à un répertoire pour les systèmes d'exploitations.

Ce qui est défini dans un espace de noms n'existe pas en dehors. Pour l'utiliser il faut utiliser le chemin d'accès :

```
int main(){
fon1() // erreur fonc1() n'existe pas
MaBiblio::fonc2() // ok
...}
```

Pour éviter d'alourdir le code, il est possible de préciser que les membres d'un espace de noms seront utilisés directement grâce aux mot clef **using**.

Il y a deux méthodes :

Accéder à tout l'espace de noms

```
using namespace MaBiblio
fonc1() // ok
fonc2() // ok
```

Préciser les fonctions à utiliser

```
using Mabiblio::fonc1( )
fonc1() // ok
fonc2() // fonc2 n'existe pas en dehors
// de l'espace de nom
```

Espaces de noms

- La plupart des fonctions et classes de la librairie standard du C++ sont déclarées dans un espace de nom
- L'espace de nom **std** contient la grande majorité des fonctions de la librairie standard
- Habituellement, il est nécessaire d'utiliser **« using namespace std »** au début de chaque programme C++ qui utilise cette librairie

Les variables

- Une variable est une abstraction d'une portion de mémoire
- Une variable est caractérisée par :

  (Nom, adresse, valeur, type, durée de vie, portée)
- L'utilisateur décide du nom et du type de la variable.
- L'emplacement de la déclaration décide de sa portée. Son adresse est déterminée pendant l'exécution et sa valeur dépend des instructions dans lesquelles elle apparaît.
- Le nommage des variables ne respecte pas de règle particulière hormis l'exclusion des espaces et des caractères accentués

Les variables

Type

Le C, et encore plus le C++, est un langage typé, chaque entité doit disposer d'un type. Faisant la différence entre le nom variable et son type.

La variable correspond a une portion de la mémoire et que sa taille sera défini par un nombre de bytes ou d'octets déterminé par le type la variable

Void	**:**	**type vide (pas de type)**
Bool	**:**	**type logique (vrais/faux) (C++)**
Char	**:**	**caractères**
int	**:**	**entiers**
float	**:**	**réels**
double	**:**	**réels en double précision**
Tableux		
Pointeurs		

2 à 4 byte!!

4 byte!!

4 à 8 byte!!

Les variables

La portée d’une variables

C++ permet de déclarer des variables n’importe où et de les initialiser dans un bloc. La porté d’une variable dépend de l’endroit ou il est placé:

- globale Les variables globales sont déclarées en dehors de toute fonction Elle est allouée et initialisé automatiquement avant l'entrée dans la fonction 'main'.

  Elle est nettoyée et libérée automatiquement après la sortie de la fonction 'main'.
- locale est visible dans le block d’instructions dans lequel elle à été déclaré et ce, à partir de sa déclaration. Elle est allouée et initialisée.

  Elle est nettoyée et libérée automatiquement, lorsqu’on sort du block d’instructions
- Dynamique est totalement contrôlée par le programmeur.

  Nous parlerons plus tard de ce type de variable.

Instructions conditionnelles

L'instruction if

Forme :

```
if (<expression entière>)
<instruction1>
else
<instruction2>
```

Mécanisme : l'<expression entière> est évaluée.

- si sa valeur est ≠ 0, l'<instruction1> est effectuée,
- si sa valeur est nulle, l'<instruction2> est effectuée.

En C++, il n'y a pas de type booléen (c'est-à-dire logique). On retiendra que :

Toute expression entière ≠ 0 (resp. « gale à 0) est considérée comme vraie (resp. fausse).

Instructions conditionnelles

Opérateurs booléens

Forme :

! : non,

&& : et,

|| : ou.

L'évaluation des expressions booléennes est une évaluation courte. Par exemple, l'expression u && v prend la valeur 0 dès que u est évalué à 0, sans que v ne soit évalué.

Opérateurs de comparaison

== : égal,

!= : différent,

< : strictement inférieur,

> : strictement supérieur,

<= : inférieur ou égal,

>= : supérieur ou égal.

Instructions conditionnelles

Exemple

On veut définir une fonction calculant le maximum de trois entiers.

```
int Max(int x, int y, int z) //
calcule le maximum de trois
entiers
{
if ((x <= z) && (y <= z))
return z;
if ((x <= y) && (z <= y))
return y;
return x;
}
```

```
int Max(int x, int y, )  // calcule le maximum de deux entiers
{
if (x < y)                return y;
else                      return x;
}
int Max(int x, int y, int z) // calcule le maximum de trois entiers
{
return Max(Max(x, y), z);
}
```

Instructions conditionnelles

L'instruction switch

Cette instruction permet un branchement conditionnel multiple.

Forme :

```
switch (<expression>)
{
case <valeur 1> :     <instructions>
...
case <valeur n> :     <instructions>
default :   <instructions>
}
```

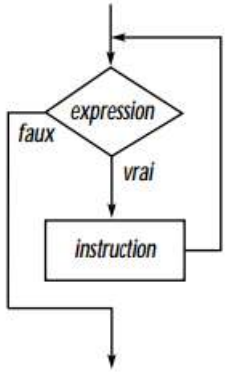
Boucles

L'instruction while

Forme :

```
while (<expression entière>)
<instruction>
```

Mécanisme : l'<expression entière> est d'abord évaluée. Tant qu'elle est vraie (c'est-`a-dire ≠ 0), l'<instruction> est effectuée.



On remarque que le test est effectué avant la boucle.

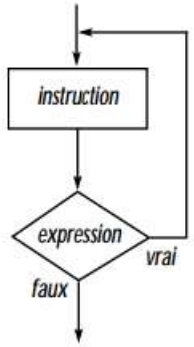
Boucles

L'instruction do ….. while

Forme :

```
do
        <instruction>
while (<expression entière>);
```

Mécanisme : l'<instruction> est effectuée , puis l'<expression entière> est évaluée. Tant qu'elle est vraie, l'<instruction> est effectuée.



On remarque que le test est effectué après la boucle.

Boucles

L'instruction for

Forme :

for (*<expression1>*; *<expression2>*; *<expression3>*)

<instruction>

Mécanisme :

<expression1>; // initialisation de la boucle
while (*<expression2>*) // test de continuation de la boucle
{
    <instruction>
    <expression3>;
}

Boucles

Exemple

Ecrivons un programme qui permettant d'entrer au clavier dix nombres entiers et qui affiche ensuite la moyenne arithmétique de ces dix nombres.

```
int n, somme = 0;
int i = 0;
cout << "Quand on le demande, tapez un entier suivi de <entrée>\n";
while (i < 10)  // boucle version while
{
cout << "Entrée ? ";
cin >> n;
somme += n;
i++;
}
cout << "La moyenne est " << somme / 10.0;
```

TP

Les pointeurs

Définitions

- Un pointeur **p** est une variable qui sert à contenir l'adresse mémoire d'une autre variable
- Si la variable **v** occupe plus d'une case mémoire, l'adresse contenue dans le pointeur **p** est celle de la première case utilisé par **v**
- La variable **v** est appelé variable pointée
- Si p contient l 'adresse de **v** alors on dit que **p** pointe sur **v**
- Le pointeur possède un type qui dépend du type des variables pointées
- Un pointeur peut contenir aussi l'adresse d'une fonction, on parlera alors de pointeur de fonctions

Les pointeurs

Déclaration

- Lorsqu'on déclare un pointeur on indique le type des variables sur les quelles il pointe

```
char  * pc;        // pc est un pointeur de caractères
int     * pi ;     // pi est un pointeur d 'entiers
float   * pf;      // pf est un pointeur de flottants
```

- Déclaration de plusieurs sur une même ligne

```
int     * p1, i1, *p2, i2; // pi est un pointeur d 'entiers
```

- p1et p2 sont des pointeurs d 'entiers*
- i1,i2 sont des entiers
- Il faut répéter les * pour chaque pointeur

Les pointeurs

Adresse d ’une variable : l ’opérateur &

- L’adresse d’une variable peut être obtenue grâce à l’opérateur **&** suivi du nom de la variable

```
int  i=5;
cout <<"valeur de i = " << i<< endl;
cout <<"adresse de i = " << &i<< endl;
```

Résultats d ’exécution:

valeur de i = 5

adresse de i = 0x0012FF78

Ce résultats dépend de la machine

Les pointeurs

Affectation d’une adresse à un pointeur

```
int  i=5;
int * pi;
pi=&i;
cout <<"valeur de i = " << i<< endl;
cout <<"adresse de i = " << &i<< endl;
cout <<« valeur de pi = " << pi<< endl;
```

Résultats d ’exécution:

valeur de i = 5

adresse de i = 0x0012FF78

valeur de pi = 0x0012FF78

Les pointeurs

Incompatibilité de types

On ne peut pas affecter à un pointeur de type T1 l'adresse d 'une variable d 'autre type T2

```
int  i=5;
char * pc;
pc=&i;
```

On peut cependant forcer la conversion.

A utiliser avec précaution ....

```
int  i=5;
char * pc;
pc= ( char *) &i;
```

Les pointeurs

Opérateur d'indirection (*)

- Cet opérateur permet d 'accéder à la valeur de la variable pointée
- Si **p** contient l'adresse de la variable **v** alors ***p** désigne la variable **v** elle même

```
int  i=5;
int * pi=&i;
 cout <<"valeur de i = " << i<< endl;
cout <<"adresse de i = " << &i<< endl;
cout <<« valeur de pi = " << pi<< endl;
cout <<"valeur de *pi = " << *pi<< endl;
```

Résultats d 'exécution:

valeur de i = 5

adresse de i = 0x0012FF78

valeur de pi = 0x0012FF78

valeur de *pi =5

Les pointeurs

Pointeur Void

- Les pointeurs void sont un type particulier de pointeur
- Ils peuvent pointer sur une variable de n'importe quel type
- On ne peut pas utiliser l 'opérateur d'indirection * sur un pointeur void. Il faut d 'abord le convertir en un pointeur d 'un type donné.

```
int  a=5, b=6;
int *  pi=&a;
void *  pv = &b;          //correct
pv=pi ;                   //correct
pi =pv ;                  // !!! Erreur!!!
pi=(int*) pv              //correct
 cout <<*pv ;             // !!! Erreur!!!
cout <<*((int *) pv );    //correct
```

Les pointeurs

Incrémentation et Décrémentation d ’un pointeur

- soit p un pointeur de type T
- soit t la taille en octets du type T
- soit a la valeur de p (adresse contenue dans p)
- si on exécute p++ alors la nouvelle valeur de p sera égale à a+t
- Le même principe s ’applique pour p--

```
int   i=5; int *pi=&i;
cout <<"valeur de pi  avant incrémentation = " << pi<< endl;pi++;
cout <<"valeur de pi  après incrémentation = " << pi<< endl;
```

Résultats d ’exécution:
valeur de px avant incrémentation = 0x0012FF78
valeur de px après incrémentation = 0x0012FF80

Les pointeurs

Addition et soustraction d'un entier

- soit p un pointeur de type T
- soit t la taille en octets du type T
- soit a la valeur de p (adresse contenue dans p)
- soit n un entier
- si on exécute p+n alors la nouvelle valeur de p sera égale à a+n*t
- Le même principe s'applique pour la soustraction d 'un entier

```
int   i=5; int *pi=&i;
cout <<"valeur de pi  avant l 'addition = " << pi<< endl;pi+2;
cout <<"valeur de pi  après l 'addition = " << pi<< endl;
```

Résultats d 'exécution:
valeur de pi avant l 'addition = 0x0012FF78
valeur de pi après l 'addition = 0x0012FF80

TP

Fonctions

En C++, la partie exécutable d'un programme (c'est-à-dire celle qui comporte des instructions) n'est composée que de fonctions. Chacune de ces fonctions est destinée à effectuer une tâche précise et renvoie généralement une valeur, résultat d'un calcul.

Une fonction est caractérisée par :

- son nom,
- le type de valeur qu'elle renvoie,
- l'information qu'elle reçoit pour faire son travail (paramètres),
- l'instruction-bloc qui effectue le travail (corps de la fonction).

Fonctions

Les trois premiers éléments sont décrits dans la déclaration de la fonction.

Toute fonction doit être définie avant d'être utilisée.

Dans un fichier-source **tp.cpp**, il est possible d'utiliser une fonction qui est définie dans un autre fichier. Mais, pour que la compilation s'effectue sans encombre, il faut en principe que cette fonction soit déclarée dans **tp.cpp** (en usant au besoin de la directive **#include**) .

La définition d'une fonction se fait toujours au niveau principal. On ne peut donc pas imbriquer les fonctions les unes dans les autres, comme on le fait en Pascal.

Fonctions

Déclaration d'une fonction

Elle se fait grâce à un *prototype* de la forme suivante :

<type> <nom>(<liste de paramètres>);

Où <**type**> est le type du résultat, <**nom**> est le nom de la fonction et <**liste de paramètres**> est une ou plusieurs déclarations de variables, séparées par des virgules.

Exemples :

```
double Moyenne(double x, double y);
char LireCaractere();
void AfficherValeurs(int nombre, double valeur);
```

Fonctions

Définition d'une fonction

Elle est de la forme suivante :

<type> <nom>(<liste de paramètres formels>)

<instruction-bloc>

Exemples :

```
double Moyenne(double x, double y)
{
return (x + y) / 2.0;
}
```

```
void AfficherValeurs(int nombre, double valeur)
{
cout << '\t' << nombre << '\t' << valeur << '\n';
}
```

Fonctions

Utilisation d'une fonction

Elle se fait grâce à l'*appel* de la fonction. Cet appel est une expression de la forme : *<nom>*(*<liste d'expressions>*).

Mécanisme de l'appel :

- chaque expression de la <liste d'expressions> est évaluée,
- les valeurs ainsi obtenues sont transmises dans l'ordre aux paramètres formels,
- le corps de la fonction est ensuite exécuté,
- la valeur renvoyée par la fonction donne le résultat de l'appel.
- Si la fonction ne renvoie pas de valeur, le résultat de l'appel est de type void.

Fonctions

Utilisation d’une fonction

```
double u, v;
cout << "\nEntrez les valeurs de u et v :";
cin >> u >> v;
double m = Moyenne(u, v);
cout << "\nVoulez-vous afficher la moyenne ? ";
char reponse;
cin>> reponse;
if (reponse == ’o’)
cout << m;
```

TP

Tableaux

Un tableau est une collection indicée de variables de même type.

Forme de la déclaration :

```
<type> <nom> [<taille>];
```

où :

<type> est le type des éléments du tableau,

<nom> est le nom du tableau,

<taille> est une constante entière égale au nombre d'éléments du tableau.

Si **t** est un tableau et i une expression entière, on note t[i] l'élément d'indice i du tableau. Les éléments d'un tableau sont indicés de **0** à **taille – 1**.

Tableaux

Il est possible de déclarer des tableaux à deux indices (ou plus). Par exemple, en écrivant :

```
int M[2][3];
```

On déclare un tableau d'entiers M à deux indices, le premier indice variant entre 0 et 1, le second entre 0 et 2. On peut voir M comme une matrice d'entiers à 2 lignes et 3 colonnes. Les éléments de M se notent M[i][j].

Comme pour les variables ordinaires, on peut faire des déclarations-initialisations de tableaux:

```
int T[3] = {5, 10, 15};
char voyelle[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
int M[2][3] = {{1, 2, 3}, {3, 4, 5}}; // ou : int M[2][3] = {1, 2, 3, 3, 4, 5};
```

Tableaux

Utilisation des tableaux

On retiendra les trois règles suivantes :

- Les tableaux peuvent être passés en paramètres dans les fonctions,
- les tableaux ne peuvent être renvoyés (avec return) comme résultats de fonctions,
- l'affectation entre tableaux est interdite.

Tableaux

Exemples

Instruction qui copie le vecteur U dans V

```
for (int i = 0; i < L; i++)
V[i] = U[i];
```

Fonction qui affiche un vecteur

```
void Affiche()
{
for (int i = 0; i < L; i++)
cout << V[i]<< endl;
}
```

Chaînes de caractères

En C++, une chaîne de caractères n'est rien d'autre qu'un tableau de caractères, avec un caractère nul **'\0'** marquant la fin de la chaîne.

Exemples de déclarations :

```
char nom[20], prenom[20];
char adresse [3][40];
String test= "supmti";
```

```
prenom="adil";
strcpy(prenom,"adil");
```

Les chaînes de caractère peuvent être saisies au clavier et affichées à l'écran grâce aux objets habituels **cin** et **cout**.

Chaînes de caractères

Fonctions utilitaires sur les chaînes de caractères:

- **strlen(s)** (dans string.h) : donne la longueur de la chaîne **s**
- **strcpy(dest, source)** (dans string.h) : recopie la chaîne **source** dans **dest**
- **strcmp(s1,s2)** (dans string.h) : compare les chaînes **s1** et **s2** :

  renvoie une valeur < 0, nulle ou > 0 selon que s1 est inférieure, égale ou supérieure à s2 pour l'ordre alphabétique.
- **gets(s)** (dans stdio.h) : lit une chaîne de caractères tapée au clavier, jusqu'au premier retour à la ligne (inclus) ; les caractères lus sont rangés dans la chaîne s, sauf le retour à la ligne qui y est remplacé par '\0'.
- **tolower()** (ctype.h): convertit une lettre majuscule en minuscule
- **toupper()** (ctype.h): convertit une lettre minuscule en majuscule

Chaînes de caractères

Exemples d'utilisation des chaînes de caractères

- Fonction qui indique si une phrase est un palindrome (c'est-à-dire qu'on peut la lire aussi bien à l'endroit qu'à l'envers) :

```
int Palindrome(char s[20]) // teste si s est un palindrome
{
int i = 0, j = strlen(s) - 1;
while (s[i] == s[j] && i < j)
i++, j--;
return i >= j;
}
```

TP

Références

Une référence est une variable qui coïncide avec une autre variable.

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

<type> &<nom> = <nom var>;

Par exemple :

```
int n = 10;
int &r = n; // r est une référence sur n
```

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

```
double somme, moyenne;
double &total = somme;          // correct : total est une référence sur somme
double &valeur;                 // illégal : pas de variable à laquelle se référer
double &total = moyenne;        // illégal : on ne peut redéfinir une référence
```

Références

Une fois que la référence est déclarée alias d'une autre variable, toutes les opérations que l'on croit effectuer sur l'alias (c'est à dire la référence) sont en fait exécutées sur la variable d'origine.

```
#include <iostream>
using namespace std ;
int main() {
int i=0;
Int &valeur = i;
valeur++;
cout << i << endl ;
return 0;
}
```

valeur = i =0
valeur = 1
affiche: 1

Références

```
double i=0;
int temp = i;
int& valeur = temp;
```

- valeur est une référence à temp (mais pas i)
- Modifier valeur revient à modifier temp mais pas i car il n'y a aucun lien entre i et temp.

Passage de paramètres

Pour rappel, il y a 2 types de paramètres:

- paramètres réels: dans l'appel de fonction
- paramètres formels: dans l'entête de la fonction

En C++, 3 types de passage de paramètres:

- par valeur (existe aussi dans les langages C et Java)
- par adresse (existe aussi dans le langage C)
- par référence (uniquement dans les langages C++ et Java)

Passage de paramètres

Supposons que nous voulions définir une fonction **echange** permettant d'échanger les valeurs de deux variables entières.

Passage par valeur

```
void echange(int a, int b)
{
int aux;                    // variable auxiliaire servant pour l'échange
aux = a; a = b; b = aux;
}
```

Aucun lien entre paramètre formel et paramètre réel.

Ici l'expression echange(x, y) (où x et y sont de type int) n'a aucun effet sur x et y.

lors de l'appel de la fonction, les valeurs de x et y sont recopiées dans les variables locales a et b (on parle de passage par valeur) et ce sont les valeurs de ces variables a et b qui sont échangées.

Passage de paramètres

Supposons que nous voulions définir une fonction **echange** permettant d'échanger les valeurs de deux variables entières.

Passage par pointeurs

```
void echange(int *a, int *b) // version correcte, style C
{
int aux;
aux = *a; *a = *b; *b = aux;
}
```

Il y a un lien entre paramètre formel et paramètre réel.

Pour échanger les valeurs de x et y, il faut alors écrire echange(&x, &y), c'est-à-dire passer à la fonction les adresses de x et y (on parle de passage par adresse).

Paramètre formel a est initialisé avec le contenu de l'adresse pointant sur x.

Passage de paramètres

Supposons que nous voulions définir une fonction **echange** permettant d'échanger les valeurs de deux variables entières.

Passage par références

```
void echange(int &a, int &b) // version correcte, style C++
{
int aux;
aux = a; a = b; b = aux;
}
```

Cette fois, l'expression echange(x, y) réalise correctement l'échange des valeurs de x et y, car les variables a et b coïncident avec les variables x et y (on parle de passage par référence).

Le paramètre formel est un alias de l'emplacement mémoire du paramètre réel.

Passage d'un tableau

Par pointeur

```
#include <iostream>
using namespace std;
void fonction(int* tableau, int taille) {
for (int i=0; i<taille;i++) cout << tableau[i] << " ";
cout << endl;
}
int main() {
int tableau[2] = {1,2};
fonction(tableau,2);
return 0;
}
```

Passage d'un tableau

Par semi-référence

```
#include <iostream>
using namespace std ;
void fonction(int tableau[], int taille) {
for (int i=0; i<taille;i++) cout << tableau[i] << " ";
cout << endl;
}
int main() {
int tableau[2] = {1,2};
fonction(tableau,2);
return 0;
}
```

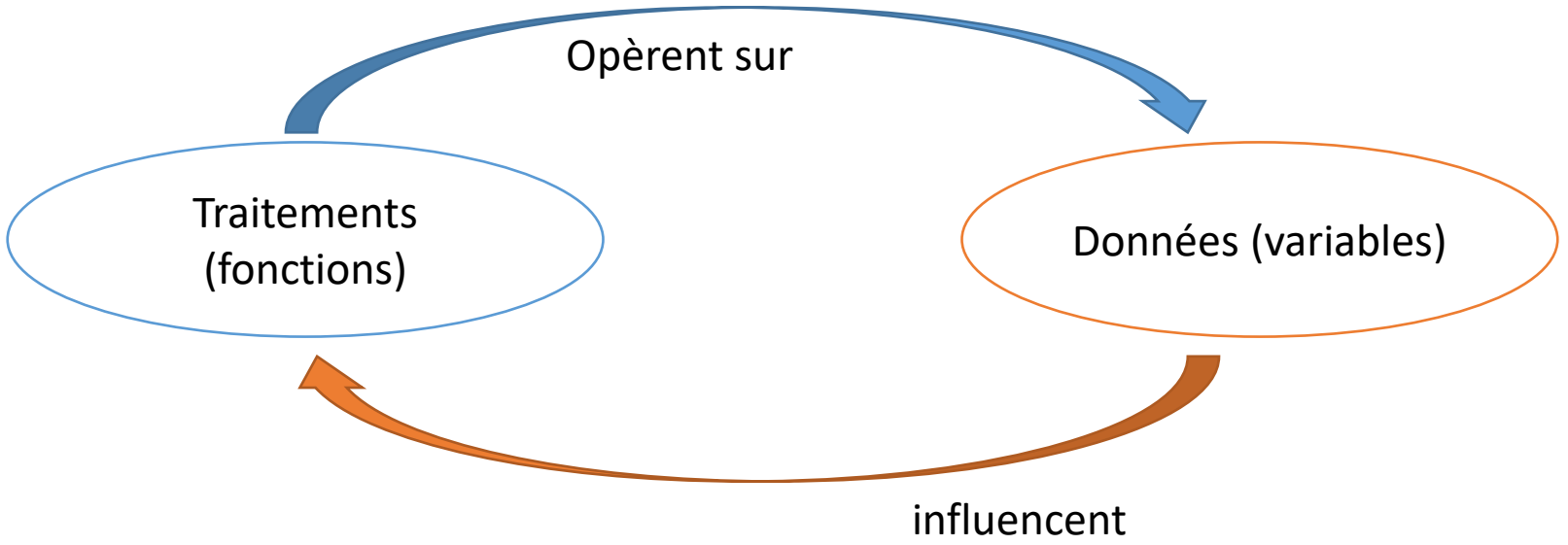
TP

Programmation Orientée Objet

Programmation Orientée Objet

Dans les programmes que vous avez écrits jusqu'à maintenant, les notions:

- De variables/types de données
- Et de traitement de ces données étaient séparées



Programmation Orientée Objet

La Programmation Orientée Objets (POO) consiste en une structuration de plus haut niveau.

Il s'agit de regrouper ensemble les données et toutes les procédures et fonctions qui permettent la gestion de ces données.

On obtient alors des entités comportant à la fois un ensemble de données et une liste de procédures et de fonctions pour manipuler ces données.

La structure ainsi obtenue est appelée : **Objet**

Programmation Orientée Objet

Un **objet** est composé de deux parties :

- Une partie statique composée de la liste des données de l'objet. On les appelle : Attributs ou Propriétés, ou encore : Données Membres.
- Une partie dynamique qui décrit le comportement ou les fonctionnalités de l'objet. Elle est constituée de l'ensemble des **procédures** et des **fonctions** qui permettent à l'utilisateur de configurer et de manipuler l'objet.

Programmation Orientée Objet

Un des objectifs principaux de la notion d'objet est d'organiser des programmes complexes grâce aux notions:

- D'encapsulation
- D'abstraction
- D'héritage
- Et de polymorphisme

Encapsulation

Un objet tel que défini plus haut est une variable caractérisée par des propriétés et des méthodes. La définition de ces propriétés et de ces méthodes devra être réalisée dans une structure de données appelée classe.

Une classe est donc le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Les données seront généralement appelées des propriétés, les fonctions qui opèrent sur ces propriétés sont appelées des méthodes.

Encapsulation

Regrouper dans le même objet informatique, les données et les traitements qui lui sont spécifiques :

- attributs: les données incluses dans un objet
- méthodes: les fonctions définies dans un objet

les objets sont définis par leurs attributs et leurs méthodes.

Encapsulation

Créer un objet depuis une classe est une opération qui s'appelle l'instanciation. L'objet ainsi créé pourra être appelé aussi : **instance**.

Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type de données et variable.

Pour utiliser une classe il faut créer une instance de cette classe (appelée aussi objet).

Programmation Orientée Objet

Déclaration d'une classe :

La déclaration est faite dans un fichier d'entête avec l'extension « .h »

```
class NomDeClasse {
 Propriétés de la classe
 Méthodes de la classe
} ;
```

Programmation Orientée Objet

Exemple:

Note .h

```
class Note {
private :
double value;
public :
void setValue(double v);
double getValue();
void print();
};
```

Visibilité des membres

3 mots clés sont utilisés pour définir un niveau de visibilité donné à un membre de la classe :

private, public et protected.

- Le mot clé **private** permet de rendre un membre privé.
- Le mot clé **public** est appliqué à des membres (en principe seulement des méthodes) pour les rendre accessibles depuis l'extérieur de la classe.
- Le mot clé **protected** offre un niveau de visibilité situé entre les 2 niveaux précédant. Des membres « protected » sont des membres inaccessibles en dehors de leur classe sauf dans des classes filles

Bonne pratique :
- Données privées
- Méthodes publiques

Définition des méthodes de la classe:

L'implémentation des méthodes préalablement déclarées dans un fichier d'entête est réalisée dans un fichier source .cpp selon la syntaxe suivante :

```
#include "Fichier d'entête.h"
Type NomDeClasse::nomDeMethode1(...) {
... }
type NomDeClasse::nomDeMethode2(...) {
... }
```

Programmation Orientée Objet

Exemple:

Note.cpp

```
#include "Note.h"
void Note::setValue(double v) {
if (v >= 0 && v <= 20) {
 value = v;
 }
}
double Note::getValue() {
return value;
}
void Note::print() {
 cout…..
}
```

Accesseurs

Parmi les méthodes pouvant assurer l'accès contrôlé aux propriétés qui sont habituellement privées, il est à définir des méthodes d'introduction et de récupération de données. Ces méthodes sont appelées des **Accesseurs**, ou encore **getters** et **setters**

```
void setX(type x) ;
type getX() ;
```

Exercice 1:

Réaliser une classe RTemp permetant de manipuler la température. On prévoira :

- Un constructeur
- Une fonction incr pour incrémenter la température
- Une fonction decr pour décrémenter la température
- Une fonction valeur pour récupérer la température

Instanciation

Un objet d'une classe est appelé aussi « **instance** ». Il existe deux types d'objets en C++ : Les objets « **statiques** » et les objets « **dynamiques** ». L'opération de création d'une instance de classe est appelée « **Instanciation** ».

Instanciation : Objets statiques

La déclaration d'un objet statique est simplement réalisée de la manière suivante :

NomDeClasse nomD'objet ;

Suite à cette déclaration l'instance est automatiquement créée et est prête à usage. L'accès aux membres (publiques) est réalisé à l'aide de l'opérateur « . »

```
void fonc() {
        Note n1; // Déclaration d'un objet
         //n1.value = 25; Accès impossible
        n1.setValue(17);
        n1.print();
        n1.setValue(25);
        cout << "La note après modification : " n1.getValue() <<endl;
}
```

Instanciation : Objets dynamiques

Dans ce cas, il est à séparer entre la déclaration est l'instanciation. La déclaration permet juste de définir un pointeur sur une instance non encore créée. La création ou l'instanciation soit être réalisée explicitement à l'aide de l'opérateur new qui se charge de créer une instance (ou un objet) de la classe et de l'associer à la variable (ou pointeur).

objet = new NomDeClasse() ;

L'accès aux membres publiques est dans ce cas réalisé à l'aide de l'opérateur « -> ».

```
void exp03() {
            Note *n1; // Déclaration :
            // Instanciation :
            n1 = new Note();
            n1->setValue(18);
            n1->print();
            delete n1;
}
```

L’objet « this »

Il existe un objet dynamique (pointeur) prédéfini « this » pouvant être utilisé à l’intérieur d’une classe, permettant de référencer à l’aide de la notation « this->… » les membres (propriétés ou méthodes) de l’objet courant. L’utilisation de « this » permettra par exemple de distinguer entre une propriété et une variable locale ou un paramètre.

```
void Note::setValue(double value) {
            if (value >= 0 && value <= 20) {
            this->value = value;
 }
 }
```

Surcharge ou Surdéfinition

Il s'agit de définir une même méthode plusieurs fois dans la même classe. Dans ce cas il serait nécessaire que chaque définition soit faite avec une signature différente.

```
void setDate(Date birthday);
void setDate(int day, int month, int year);
```

Héritage

L'héritage consiste à développer de nouvelles classes issues de classes existantes. Il s'agit d'étendre les fonctionnalités de classes déjà définies en ajoutant de nouvelles propriétés et méthodes.

```
class Fille : public Mère {
};
```

Cette simple écriture permettra de créer une nouvelle classe nommée : Classe **Fille** ou classe **Dérivée**, depuis une classe existante nommée : classe **Mère, classe de base** ou encore **super classe**.

La classe fille sera alors une extension de la classe mère. En plus, tout objet de la classe fille pourra jouer le même rôle que celui des objets de la classe mère, avec en plus des possibilités supplémentaires.

Héritage

```
class Point {
private:
        int x, y;
public:
        Point(void);
        Point(int x, int y);
        void setX(int x);
        void setY(int y);
        int getX();
        int getY();
        void print();
        ~Point(void);
};
```

```
        Point p1(20, 30);
        p1.print();
```

```
class ExtendedPoint : public Point {
private:
        Int color;
public:
        ExtendedPoint(void);
        ExtendedPoint(int x, int y, int color);
        void setColor(int color);
        int getColor();
        ~ExtendedPoint(void);
};
```

```
        ExtendedPoint p;
        p.setX(20);
        p.setY(30);
        p.print();
```

Programmation Orientée Objet

Une méthode définie dans une classe mère peut ne plus être valable pour une classe fille.

```
void Point::print() {
cout << "Point("<< x<<","<< y<<") " <<endl;
 }
```

Cette méthode peut être invoquée sur des objets de la classe fille « **ExtendedPoint** » comme dans l'exemple précédent. Mais le résultat sera comme suit : Point(50, 60)

L'**ExtendedPoint** est affiché comme un Point.

Pour cela, il sera nécessaire de reprendre la définition de la méthode « **print** » dans la classe « **ExtendedPoint** ». Il s'agit d'un autre concept de la programmation orientée objet qu'on appelle « **Redéfinition** ».

Surcharge des opérateurs

```
void Point::operator =(Point p) {
this->x = p.getX() * 2;
this->y = p.getY() * 2;
}
```

```
void test() {
 Point p1(20, 30);
 Point p2;
 p2 = p1;
 p2.print();
}
```

Surcharge des opérateurs

```
Point Point::operator +(Point p) {
 Point s(this->x + p.getX(), this->y + p.getY());
return s;
}
```

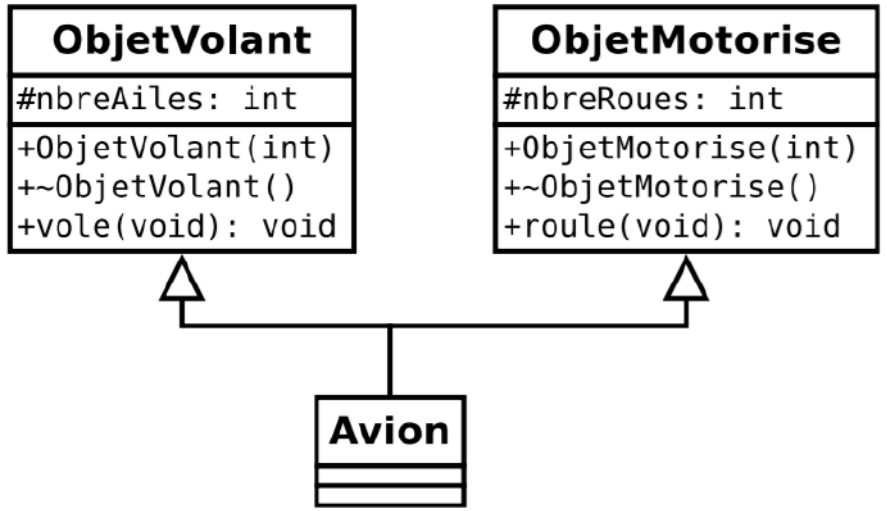
```
void test2() {
 Point p1(20, 30);
 Point p2(25, 14);
 Point p3;
 p3 = p1 + p2;
 p3.print();
}
```

L'Héritage Multiple

En C++, une classe peut hériter de plusieurs classes mères on parle **d'héritage multiple**

- la classe dérivée hérite des attributs et méthodes de toutes ses classes mères
- **attention** : ce n'est pas possible dans tous les langages orientés objet

Exemple



L'Héritage Multiple

- La définition de l'héritage multiple se fait de manière similaire à celle de l'héritage simple.

```
class Fille : public Mère1, public Mère2, ….. , public Mèren{
                };
```

L'Héritage Multiple

- Exemple de la classe Avion

```
class ObjetVolant {
protected:
        int nbreAiles;
public:
        ObjetVolant (int n)
                {
                nbreAiles = n;
                cout << "Demarrage d'un objet volant a " << nbreAiles << " ailes" << endl; }
        virtual ~ ObjetVolant ()
                {
                cout << "Arret de l'objet volant a " << nbreAiles << " ailes" << endl;}
        void vole(){
                cout << "ca vole ..." <<  endl; }
};
```

L'Héritage Multiple

- Exemple de la classe Avion

```
class ObjetMotorise {
protected:
        int nbreRoues;
public:
        ObjetMotorise (int n)
                {
                nbreRoues = n;
                cout << "Demarrage d'un objet motorise a " << nbreRoues << " roues" <<  endl;}
        virtual ~ ObjetMotorise ()
                {
                cout << "Arret de l'objet motorise a " << nbreRoues << " roues" << endl;}
        void roule(){
                cout << " ca roule..." <<  endl; }
};
```

L'Héritage Multiple

- Exemple de la classe Avion

- la classe Avion hérite des attributs (**nbreAiles** et **nbreRoues**) et des méthodes (**vole** et **roule**) des classes **ObjetVolant** et **ObjetRoulant**
- pour une classe dérivée donnée, il n'y a pas de restrictions sur le nombres de classes mères
- l'ordre des déclarations des classes mères a une incidence sur les appels des constructeurs et destructeurs

L'Héritage Multiple

- Appel aux constructeurs et destructeurs

  - De la même manière que pour l'héritage simple, pour l'héritage multiple, l'initialisation des instances des classes de bases se fait sur le même modèle que l'initialisation des membres constants avant l'appel du constructeur

```
class Avion : public ObjetVolant , public ObjetMotorise
{
   public:
      Avion();
      virtual ~Avion();
   protected:
   private:
};
```

L'Héritage Multiple

- Ambiguïté sur les attributs et les méthodes

  - Une ambiguïté peut se produire lorsque le nom d'un attribut ou d'une méthode est le même dans deux classes mères différentes
  - Par exemple : on ajoute la méthode roule à la classe **ObjetVolant**.
  - Solution 1: utilisation de l'opérateur de résolution de portée **Avion a; a. ObjetVolant :: roule () ;**
  - Solution 2: redéfinition de la méthode dans la classe dérivée
  - Solution 3: indication explicite de quelle méthode appelée **using ObjetVolant :: roule;**

STL (Standard Template Library)

La bibliothèque STL (Standard Template Library) de C++ est un ensemble de classes et d'algorithmes génériques qui simplifient la gestion des structures de données et des algorithmes.

STL (Standard Template Library)

Les Conteneurs

Les conteneurs STL permettent de stocker, d'organiser et de manipuler des collections de données. Voici les catégories principales :

std::vector

Conteneur dynamique : Taille ajustable, accès rapide par index.

Méthodes importantes :

push_back(), pop_back(), size(), clear(), at()

STL (Standard Template Library)

std::vector

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = {1, 2, 3};
    v.push_back(4); // Ajouter un élément à la fin
    v.pop_back();   // Supprimer le dernier élément

    std::cout << "Taille : " << v.size() << std::endl;
    for (int val : v) {
        std::cout << val << " "; // Résultat : 1 2 3
    }

    return 0;
}
```

STL (Standard Template Library)

std::list

Liste doublement chaînée : Efficace pour les insertions/suppressions.

Méthodes importantes :

push_front(), push_back(), pop_front(), pop_back(), reverse()

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> l = {1, 2, 3};
    l.push_front(0); // Ajouter au début
    l.reverse();     // Inverser la liste

    for (int val : l) {
        std::cout << val << " "; // Résultat : 3 2 1 0
    }

    return 0;
}
```

STL (Standard Template Library)

std::set

Ensemble ordonné sans doublons.

Méthodes importantes :

insert(), find(), erase()

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> s = {3, 1, 4};
    s.insert(2);    // Ajouter un élément
    s.insert(4);    // Ignoré car 4 existe déjà

    for (int val : s) {
        std::cout << val << " "; // Résultat : 1 2 3 4
    }

    return 0;
}
```

STL (Standard Template Library)

std::map

Table associative : Stocke des paires clé-valeur.

Méthodes importantes :

insert(), find(), erase(), at()

```cpp
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> m;
    m["Alice"] = 25;
    m["Bob"] = 30;

    std::cout << "Alice a " << m["Alice"] << " ans." << std::endl;

    return 0;
}
```

STL (Standard Template Library)

std::queue

File d'attente (FIFO) : Ajout à la fin, suppression au début.

Méthodes importantes :

push(), pop(), front(), back()

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);

    std::cout << "Premier élément : " << q.front() << std::endl;
    q.pop();
    std::cout << "Après pop, premier élément : " << q.front() << std::endl;

    return 0;
}
```

STL (Standard Template Library)

std::priority_queue

File de priorité : Les éléments sont triés selon leur priorité.

Méthodes importantes :

push(), pop(), top()

```cpp
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    pq.push(10);
    pq.push(20);
    pq.push(15);

    std::cout << "Élément avec la plus haute priorité : " << pq.top() << std::endl;

    return 0;
}
```

STL (Standard Template Library)

std::stack

Pile (LIFO) : Dernier arrivé, premier servi.

Méthodes importantes :

push(), pop(), top()

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    s.push(1);
    s.push(2);

    std::cout << "Sommet de la pile : " << s.top() << std::endl;
    s.pop();
    std::cout << "Après pop, sommet : " << s.top() << std::endl;

    return 0;
}
```

STL (Standard Template Library)

Les Itérateurs

Les itérateurs permettent de parcourir les conteneurs.

Exemple : Itérateur avec std::vector

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = {1, 2, 3};
    std::vector<int>::iterator it;

    for (it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " "; // Résultat : 1 2 3
    }

    return 0;
}
```

STL (Standard Template Library)

Les Algorithmes

La STL fournit plusieurs fonctions utilitaires dans l'en-tête <algorithm> .

std::sort

Trie un conteneur

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {5, 2, 8, 1};
    std::sort(v.begin(), v.end());

    for (int val : v) {
        std::cout << val << " "; // Résultat : 1 2 5 8
    }

    return 0;
}
```

STL (Standard Template Library)

Les Algorithmes

La STL fournit plusieurs fonctions utilitaires dans l'en-t

std::find

Cherche un élément

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3};
    auto it = std::find(v.begin(), v.end(), 2);

    if (it != v.end()) {
        std::cout << "Élément trouvé : " << *it << std::endl;
    } else {
        std::cout << "Élément non trouvé." << std::endl;
    }

    return 0;
}
```

STL (Standard Template Library)

Les Algorithmes

La STL fournit plusieurs fonctions utilitaires dans l'en-tête

std::reverse

Inverse l'ordre des éléments

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3};
    std::reverse(v.begin(), v.end());

    for (int val : v) {
        std::cout << val << " "; // Résultat : 3 2 1
    }

    return 0;
}
```

STL (Standard Template Library)

Les Algorithmes

La STL fournit plusieurs fonctions utilitaires dans l'en-tête <algorithm>

std::accumulate

Calcule la somme des éléments

```cpp
#include <iostream>
#include <vector>
#include <numeric>

int main() {
    std::vector<int> v = {1, 2, 3};
    int sum = std::accumulate(v.begin(), v.end(), 0);

    std::cout << "Somme : " << sum << std::endl; // Résultat : 6

    return 0;
}
```

STL (Standard Template Library)

Les Algorithmes

La STL fournit plusieurs fonctions utilitaires dans l'en-tête <algorithm> .

std::count

Compte les occurrences d'un élément

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 2, 3};
    int count = std::count(v.begin(), v.end(), 2);

    std::cout << "Nombre d'occurrences de 2 : " << count << std::endl;

    return 0;
}
```