C++

Introductions générale

Les bibliothéques de bases à inclure :

#include <iostream>
#include <string> *Si on va travailler avec des chaines*
Using namespace std;
System("PAUSE"); *Remplacer getch en C*

Notions d'entrée sortie :

Affichage
Cout<<"Message"<<endl;
Cout<<variable<<endl;
Lecture à partir du clavier
Cin>>variable1>>variable2;

Les chaines de caractéres :

1ére Méthode (Ancienne) :
Déclaration : char * ch = "bonjour";
Affichage : cout<<ch<<endl<<
Affichage de l'adresse : cout<<(void*)ch<<endl;

2éme Méthode (Nouvelle) :
Il faut importer la bib <string>
Déclaration : string ch ;

Les déclarations des variables ne doivent plus etre forcément au début du programme

Passage par référence :

- Trés similaire au passage par adresse
- Si on déclare une variable "int x" alors "int &ref = x" est la référence sur x
- La référence doit etre toujours efféctué sur une variable et non pas sur une constante
- Cette notion est principalement utilisé dans les fonctions qui agissent sur les variables :

Passage par adresse	Passage par réference

Void permuter (int *x, int*y)
Int aux;
Aux = *x;
*x = *y;
*y = aux;
	Void permuter (int &x, int &y)
	Int aux;
	Aux = x;
	x = y;
	y = aux;

On peut attribuer des variables par défaut aux paramétres des fonctions (identique a Python)

Surdéfinition des fonctions :

S'il existe deux fonctions ayant le meme nom alors:

Conversions implicites:
char➔ short ➔ int ➔ long
float➔ double

```
void fct (int) ;          // fct1
void fct (double) ;       //fct 2

char c ;  float y ;

fct(c); // appelle fct1, après conversion de c en int
fct (y); // appelle fct2, après conversion de y en double
```

Opérateurs *new* et *delete* et notions de pointeurs:

- *New :*
  - On déclare un pointeur : int* pointer
  - On l'associe avec *new* : pointer = new int
  - Si on va pointer sur un tableau on utilise : pointer = new int[taille];
- *Delete : Libérer l'espace mémoire prise par le pointeur :*
  - Delete pointeur;
  - S'il s'agit d'un tableau, on utilise : delete []pointer
- *Les variables déclarés avec new sont utilisés comme des pointeurs.*

Déclaration des fonctions inline :

En C, les fonctions sont déclarés dans un fichier séparé .cpp avec leurs prototypes dans un fichier .h.

Notion d'espace de noms : namespace nom {} / using namespace nom;
Si on n'utilise pas *using namespace std,* on doit faire appel aux fonctions *cin* et *cout* de cette façon :
Std::cin / std::cout
L'opérateur :: s'appelle opérateur de résolution de portée

Programmation orientée objet (Partie 1)

Inroduction:

- Une structure est une classe don't aucun des données n'a été encapsulé.
- Si on souhaite ajouter une méthode a une structure, on ajoute le prototype dans la définition de la structure, et on définit la méthode sous cette façon :
  - NomStructure::nomMethode(int a,int b…){}
- En c++ on peut supprimer le mot *struct* pour déclarer une variable d'une structure:
  - Exple *: struct point a; => point a;*

Déclaration d'une classe :

- Class nomClasse {}
- Par défaut tous les attirubts et méthodes sont privés.
- Si un attribut est privé (encapsulé) on ne peut pas l'accéder directement (nomVar.nomAttribute)
- Pour déclarer un ensemble d'attributs publics on utlise *public :*

Fonctions getters et setters :

- Si on déclare les attributs en privé, il nous faut des fonctions appelés getters et setters.
- Par convention les méhtodes sont déclarés sous cette forme :
    - (Type de l'attribut) getNomAttribute()
    - Void setNomAttriubte()

Création d'un objet :

- Par allocation :
    - nomClass* variable = new nomClass()
- Par déclaration :
    - nomClass variable;

On peut affecter deux instances de la meme classe, de telle façon touts les attributs vont etre copiés de l'un à l'autre.
Il faut faire attention lorsque l'un des attributs est un pointeur sur un tableau.
Dans ce cas Il faut faire une nouvelle allocation pour que les 2 instances ne pointent pas sur le meme tableau.

Constructeur et destructeur :

- Le constructeur est une méthode qui posséde le meme nom de la classe, et qui est appelé automatiquement à chaque instansciation
- Le destructeur est une méthode de la forme ~nomClasse

Attrbuts statiques :

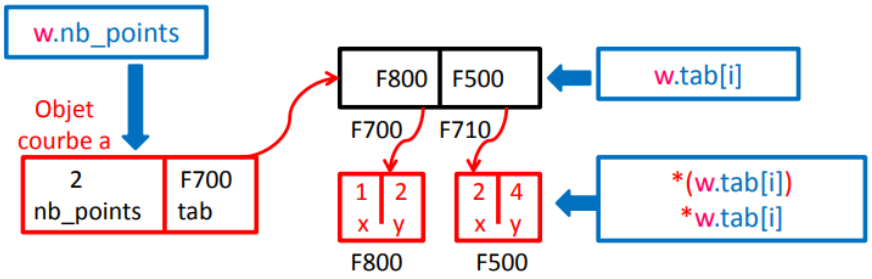
- On utilise le mot *static* pour déclarer des attributs statiques.
- Ils sont partagés entre tous les objets de la classe et sont uniques.
    - *Similaire aux attributs de classe étudiés avant.*
- Ils peuvent etre appeles meme si aucun objet n'a été crée:
    - nomClasse::nomAttribut();
- Exple :
    - Class nomClasse{
        - Static int nb_objets;
    - }
    - *L'ecriture static int nb_objets* = 0 dans la classe est ***fausse.***
    - *Il faut impérativement les initialiser dans un fichier.cpp avec l'appel :*
      Int nomClasse::nb_objets = 0;

Protections des inclusions multiples :
#pragma once

Constructeur de recopie:
On utilise un constructeur de recopie lorsque on posséde des parties dynamiques au sein de la fonction, il est déclaré comme suit:

nomClasse::nomClasse(const nomClasse &instance)

courbe::courbe(const courbe &w)




```
courbe::courbe(const courbe &w)
{
    nb_points=w.nb_points;
    tab=new point*[nb_points];
    for(int i=0; i<nb_points; i++)
        tab[i]=new point(*w.tab[i]);
}
```

Création d'un objet :

Par déclaration :

Appel du constructeur :

nomClasse obj(parametres constructeurs)

Constructeur de recopie:

nomClasse obj = Ancien obj (Déclaration + Initialisation)

Par allocation :

Appel du constructeur :

nomClasse * var = new nomClasse(parametres constructeurs)

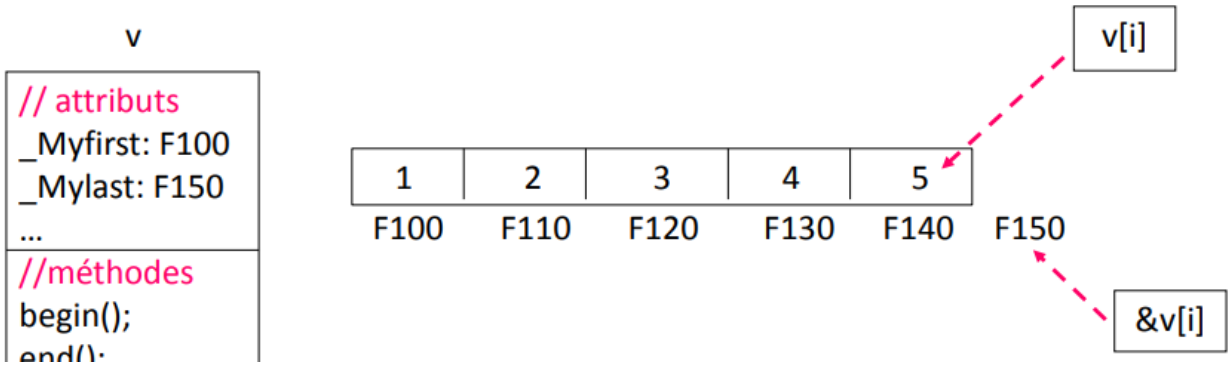
Constructeur de recopie :

nomClasse * var = new nomClasse(ancienObjet)

Tableau d'objets :

On utilise géneralement un vector :

v.push_back(val);	ajoute la valeur val à la fin du vecteur v.
v.size();	retourne le nombre d'éléments dans le vecteur v.
v[i]	l'élément d'indice i dans le vecteur v (1er élément d'indice 0).
v.erase(v.begin()+i);	supprime l'élément d'indice i
v.insert(v.begin()+i, val);	insère la valeur val à la position d'indice i
v.pop_back();	supprime le dernier élément
v.clear();	supprime tous les éléments du vecteur (vider le vecteur)



Héritage

Syntaxe :

Class classFille : public classeMere{}

Redéfinition VS Surdéfinition :

Sifnature d'une methode :

void afficher(string ="");

Redéfinition : Dans la classe dérivée on redefinit une fonction de la classe mere tout en gardant la meme signature
Surdéfinition : On surcharge une méthode en changeant sa signature.

Controle d'accées :
Private -> Protected -> Public

		Statut des membres de la classe de base		
		public	protected	private
Mode de dérivation	Public	public	protected	Private
	Protected	protected	protected	private
	Private	private	private	private

Typage :

```
void main()
{
    point a(11,11);
    pointColore b(22,22,22);
    pointColoreMasse c(33,33,33,33);
    // conversion d'un type dérivé en un type de base
    a=b; //OK
    b=c; //OK

    // conversion d'un type de base en un type dérivé
    b=a;  //ERREUR
    c=b;  //ERREUR
    system("PAUSE");
}
```

- a=b; **est légale**. Elle entraîne une **conversion** de *b* dans le type *point* et l'affectation du résultat à *a*. ➔ Cette conversion revient à ne conserver de *b* que ce qui est du type *point,* elle n'entraîne pas la création d'un nouvel objet.

Solution: Typage dynamique

C++ permet d'effectuer l'identification d'un objet au moment de l'exécution (**et non pas à la compilation**)

Cela nécessitera l'emploi de **fonctions virtuelles.**

Lorsqu'une fonction est redéfinie dans une classe dérivée, elle doit être virtuelle dans la classe de base.

Lorsqu'une classe comporte une fonction virtuelle, elle doit rendre son destructeur virtuel.

une **fonction virtuelle** est une fonction définie dans une classe et qui est destinée à être redéfinie dans les classes dérivées.

Polymorphisme :

Polymorphisme statique : surdéfiniton

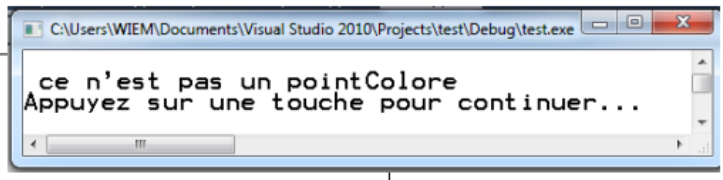
Polymorphisme dynamique : redéfinition

Typeinfo:

#include <typeinfo>

Identifier le type d'un objet

```
#include<typeinfo>
void main()
{
    pointColoreMasse a;
    if(typeid(a)== typeid(pointColore))
        cout<<"\n c'est un pointColore "<<endl;
    else cout<<"\n ce n'est pas un pointColore "<<endl;
    system("PAUSE");
}
```



Comparaison des types de deux objets

```
#include<typeinfo>
void main()
{
    pointColoreMasse a;
    point b;
    if (typeid(a)== typeid(b))
        cout<<"\n meme type "<<endl;
    else cout<<"\n type different"<<endl;
    system("PAUSE");
```

Static_cast:

static_cast <nomType>(expr)

Cet opérateur est utilisé pour effectuer des conversions qui sont résolues à la compilation.

Son utilisation :

- changement de type d'un pointeur d'une classe de base en un pointeur d'une classe dérivée(point* ➔ pointColore*).
- Changement de type d'un objet d'un type de base en un type dérivé (point ➔ pointColore)

Exple :

```
courbe::courbe(const courbe &w)
{
    point *q;
    for(int i=0; i<w.tab.size(); i++)
    {
        if (typeid(*w.tab[i])==typeid(point))
            q=new point(*w.tab[i]);

        else if(typeid(*w.tab[i])==typeid(pointColore))
            q=new pointColore( static_cast<const pointColore&>(*w.tab[i]));

        else if(typeid(*w.tab[i])==typeid(pointColoreMasse))
            q=new pointColoreMasse( static_cast<const pointColoreMasse&>(*w.tab[i]));

        tab.push_back(q);
    }
}
```

Recopie et héritage :

```
etudiant_salarie::etudiant_salarie(const etudiant_salarie &w):etudiant(w)
{
// la copie de nb_notes et les notes se fait par l'appel: etudiant(w)
    cout<<"\n +++ constructeur de recopie etudiant salarie +++"<<endl;
    nb_mois=w.nb_mois;
    mois= new int[nb_mois];
    for(unsigned int i=0; i<nb_mois; i++)
            mois[i]=w.mois[i];
}
```

D'une manière générale:

```
// B est une sous classe de A
B (const B & w) : A (w)
// w de type B, est converti dans le type A pour être transmis au
constructeur de recopie de A
{
        // recopie de la partie de w spécifique à B (non héritée de A)
}
```