Programmation orientée objet avec C++

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Cours 1 : Introduction à la programmation C++

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Evaluation

- 1 note de suivi de TP et TD (25%)
- 1 examen (25%)
- 1 mini projet (50%)

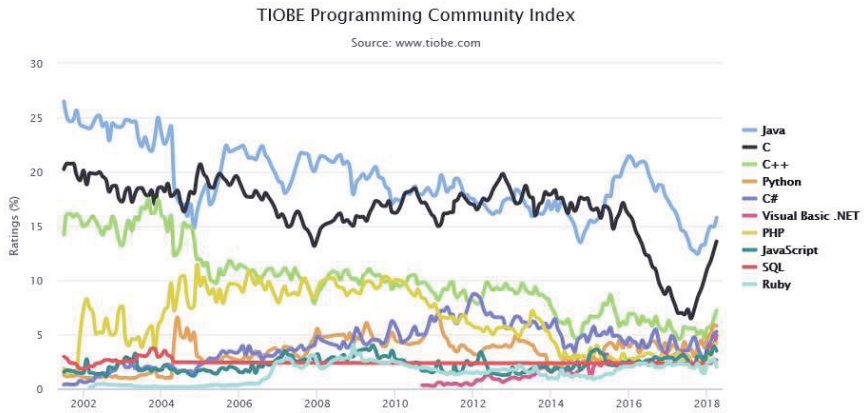
Récapitulatif de l'ensemble des cours

- Cours 1 : Introduction à la programmation C++
- Cours 2 : Les classes
- Cours 3 : L'héritage
- Cours 4 : Le polymorphisme
- Cours 5 : Les interfaces graphiques

Objectifs

- Apprendre un nouveau langage : C++
- Approfondir les connaissance en algorithmique
- Exploiter l'informatique pour résoudre des problèmes mathématiques
- Comprendre le modèle objet
- Exploiter et utiliser le modèle objet (classes, héritage, ...)
- Apprendre à créer une interface graphique
- Réaliser une application de a à z

Popularité des langages de programmation



Evolution des langages de programmation

- 19__- L'assembleur
- 1952 - A-0
- 1954 - Mark I Autocode
- 1954 - FORTRAN
- 1959 - LISP
- 1971 - Pascal
- **1972 - C**
- 1972 - SmallTalk 72

Distance entre le langage et l'architecture matérielle

- **1983 - C++** (C with classes)
- 1985 - QuickBASIC
- 1991 - Visual Basic
- 1994 - PHP
- 1995 - Java
- 1998 - C++ (ISO'98)
- 2000 - .Net, Flash
- 2002 - C#

Le langage C++

- Langage normalisé par l'ISO
- Défini dans les années 1980 (mais a évolué depuis : C++98, C++03, C++11, C++14, C++17)
- Amélioration de C qui facilite l'apprentissage pour quelqu'un qui connaît déjà C. Mais, faire du C en C++ n'est pas programmer en C++ !
- Doté d'une bibliothèque de classes et algorithmes
- Portable

C++ est presque partout :

Microsoft Windows, Office, Google Chrome, Edge, Mozilla Firefox, Oracle, Adobe Photoshop, MySQL, EA (FIFA), Ubisoft

Algorithme et programmation (1)

Définition : **Algorithme**

- Méthode pour résoudre un problème

Pour un problème donnée, il peut y avoir plusieurs algorithmes... ou aucun !

- Pour la plupart des problèmes intéressants, il n'existe pas aujourd'hui d'algorithme (Ex : Répartition des nombres premiers)
- Dans les problèmes qui restent, la grande majorité ont des algorithmes beaucoup trop durs pour être utilises ! (Ex : Jeu de Poker Vs Jeu d'échecs)
- On cherche des algorithmes simples, efficaces, élégants. . .

Définition : **Programmer**

- S'adresser à une machine = Ecrire des algorithmes

Algorithme et programmation (2)

Une fois l'algorithme trouvé, programmer en C++ comporte 3 phases:

- 1. Editer le programme avec votre éditeur favori
- 2. Compiler le programme
- 3. Exécuter le programme

  . . .
- 4. TESTER et DEBUGGER : retour au point 1 !

Ca peut durer assez longtemps...

Premiers code

L'inévitable hello world:

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

int main() {
	cout << "hello world !" << endl;
	return 0;
}
```

Affiche Hello world! dans la console

Les types de bases

- Les entiers : Nombres entiers signés ou non (int)
- Les flottants : Nombres à virgule (float)
- Les chaines : Chaines de caractères (string)
- Les booléens : 2 valeurs possibles True or False (bool)
- Les tableaux : Liste d'un même type (int Tab[n])
- Les énumérations : Regroupement de plusieurs constantes (enum)
- Les structures : Regroupement de plusieurs types (struct)

Les entiers

Type	Taille en octets	Plage
short	2	[-32768, 32767]
int	4	[-2147483648, 2147483647]
long (32b)	4	[-2147483648, 2147483647]
long (64b)	8	[-9223372036854775808, 9223372036854775807]
long long	8	[-9223372036854775808, 9223372036854775807]
unsigned short	2	[0, 65535]
unsigned int	4	[0, 4294967295]
unsigned long (32b)	4	[0, 4294967295]
unsigned long (64b)	8	[0, 18446744073509551615]
unsigned long long	8	[0, 18446744073509551615]

Les flottants

Type	Taille en octets	Plage
float	4	[-3.4*10^{-38}, 3.4*10^{38}]
double	8	[-1.7*10^{-308} à 1.7*10^{308}]
long double	10	[-3.4*10^{-4932} à 3.4*10^{4932}]

Déclaration d'un entier :

```
int a = 89;
```

Déclaration d'un flottant :

```
float b = 76.43;
```


Les chaines de caractères

Type	Taille en octets	Plage
char	1	[-128, 127]

Pour stocker du texte 2 solutions :

Un tableau de char :

```
char mot[10] = "bonjour";
```

Le type string :

```
#include <string>
string mot = "bonjour";
```


Les booléens

Type	Taille en octets	Plage
bool	4	[-2147483648, 2147483647]

Déclaration d'un booléen :

```
bool var = true;
```

Déclaration d'un tableau de 2 booléen :

```
bool Tab[2] = { true, false };
cout << Tab[0] << endl;
```

Puisque, `Tab[0] = true` le résultat affiché est 1

Les énumérations

Syntaxe d'une énumération :

```
enum JourTravail { lundi, mardi, dimanche };
JourTravail jour = dimanche;
cout << jour << endl;
```

- La variable jour ne peut prendre que 3 valeurs
- Le résultat qui s'affiche est l'indice de la variable (= 2)

Autre exemple :

```
enum SalaireJourTravail { lundi = 200, mardi = 200, dimanche = 400 };
```

Les structures

Syntaxe d'une structure :

```
struct Point { double x; double y; };
Point p;
p.x = 2.0;
p.y = 3.0;
cout << p.x << endl;
```

- La variable point contient 2 variables de type double
- La copie d'une structure se fait facilement avec l'opérateur =

Les opérateurs

- Opérateurs arithmétiques

*, +, -, / (division entière et réelle), % (modulo)

- Opérateurs de comparaison

< (inférieur), <= (inférieur ou égal), == (égal), > (supérieur), >= (supérieur ou égal) et != (différent)

- Opérateurs booléens

&& (ET), || (OU), !(NON)

Par exemple : `(x < 12) && (z != 4)`

Les instructions (1)

- La condition IF

```
if (i == 5)
      { i = 0; }
```

- La boucle FOR

```
for (int i = 0; i < 20; i++)
      { cout << i << endl; }
```

- L'instruction BREAK

```
for (int i = 0; i < 20; i++)
      { cout << i << endl;
      if (i == 5) { break; } }
```


Les instructions (2)

- La boucle WHILE

```
while (i < 20)
      { cout << i << endl;
      i = i + 1; }
```

- La boucle DO WHILE

```
do    { cout << i << endl;
      i = i + 1; }
      while (i < 20);
```

Comportement quasi identique de WHILE et de DO WHILE

Les entrées sorties

- Afficher à l'écran

```
cout << expr1 << … << exprn;
```

- Lire au clavier

```
cin >> expr1 >> … >> exprn;
```

`cout` désigne le flot de sortie, il affiche du texte ou des variables notament grace à l'opérateur `<<`

`cin` désigne le flot d'entrée standard, on lui associe des variables grace à l'opérateur `>>` Les espaces, les tabulations et les entrées marquent le passage d'une variable à une autre

Evaluation des expressions

- `y = x++;  // y = x + 1`
- `y = ++x;  // y = x`
- `x % y`
- `if ( e1 && e2 ) { … }  // l'expression e2 n'est évaluée que si e1 est true`
- `if (e1 || e2 ) { … }  // l'expression e2 n'est évaluée que si e1 est false`

```
double i = 3, j = 2, k = 3.5;
int r;
r = (i / j) * k; // r = 5
```

```
int i = 3, j = 2, r;
double k = 3.5;
r = (i / j) * k; // r = 3
```

Remarque : Si une expression mélange plusieurs types, c'est le type le plus large qui est utilisé

Les pointeurs (1)

- Les pointeurs permettent d'accéder rapidement à des zones mémoires de façon linéaire
  L'inconvénient des pointeurs et qu'ils :
  - Nécessitent de maîtriser la taille de ses données
  - Peuvent endommager la mémoire de l'ordinateur

- Exemple :

```
int a = 25;  // Déclaration d'un entier a
int b = 10;  // Déclaration d'un entier b
int *p = &a; // Déclaration d'un pointeur sur a
*p = 7;      // Equivalent à a = 7
```

Adresse	Contenu
0x01	25 (a)
0x02	10 (b)
0x03	0x01 (p)

Les pointeurs (2)

- Deux codes source équivalents réalisant la somme des données avec et sans pointeurs :

```
int sum, tableau[256];
for (int i = 0; i<256; i++)
{ sum = sum + tableau[i]; }
```

```
int sum, tableau[256];
int *p;
p = tableau;  // p = &tableau[0]
for (int i = 0; i<256; i++)
      {sum = sum + *(p++);}
```

- Le code avec pointeurs présente un gain en performance

Les fonctions (1)

- Syntaxe d'une fonction

type fonction (type parametre1, …, type parametren) {

…

}

  - Si une fonction renvoie un résultat, il doit y avoir une instruction return
  - Si non le type de la fonction est void

- Exemple : La fonction max

```
int fonction(int a, int b) {
      if (a > b) { return a; }
      else { return b; }
}
```


Les fonctions (2)

- Exemple : La fonction permutation

```
void permutation(int &a, int &b) {
      int aux = b;
      b = a;
      a = aux; }

int main() {
      int x = 5, y = 10;
      permutation(x, y);
      cout << " x = " << x << endl ;
      return 0;}
```


Application (1)

- Le nombre e est considéré parmi les nombres les plus importants en mathématiques
- Euler (1707 – 1783) découvre une nouvelle façon d'exprimer e en fraction continue simple
- Les premiers chiffres de e sont :

e = 2.71828182845904523536028747135 27

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

$$e = \sum_{n=0}^{\infty}\frac{1}{n!} = \frac{1}{1}+\frac{1}{1}+\frac{1}{1\times 2}+\frac{1}{1\times 2\times 3}+\cdots$$

$$e = 2+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{1+\ddots}}}}}}}$$

$$e = [2;1,2,1,1,4,1,1,6,1,\ldots,1,2n,1,\ldots]$$

Application (2)

- Calculons de manière itérative les deux premières formules
- Calculons de manière récursive la fraction continue simple

Première formule : $e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$

```
#include <sstream>
int main() {
      double n = 100; // précision
      double formule1 = pow(1 + 1 / n, n); // (1 + 1/n)^n
      cout << "La valeur de e = " << setprecision(20) << formule1 << endl;
      return 0;
}
```

Application (3)

Deuxième formule : $e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1\times2} + \frac{1}{1\times2\times3} + \cdots$

```
int main() {
	double n = 100, formule2 = 0;
	for (int i = 1; i < n; i++) {
		double var = 1;
		for (double k = 1; k < i; k++)
			{var = var * 1/k;} // (1 * 1/2 * ... * 1/k)
		formule2 = formule2 + var;}
	cout << "La valeur de e = " << setprecision(20) << formule2 << endl;
	return 0;
}
```


Application (4)

Troisième formule :

$$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}}}}}}$$

```
double f(double var, double i) {
if (i > 1) { i--;
return f(2 * i + 1 / (1 + 1 / (1 + 1 / var)), i); }
else { return 2 + 1 / (1 + 1 / var); }
}
int main() {
	double formule3 = f(1, 100); // Valeur de départ 1, Précision 100
	cout << "La valeur de e = " << setprecision(20) << formule3 << endl;
	return 0;
}
```


Cours 2 : Les classes

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Principaux axes du cours

- Classes et objets
- Champs et méthodes
- Constructeurs et destructeurs
- Constructeurs par copie
- Fonctions amies
- Surcharge d'opérateurs

Les classes et les objets

Définition : **Classe**

- Sert à regrouper les données
- Sert à associer les fonctions aux objets de la classe
- Permet de restreindre l'accès à certaines données

Définition : **Objet**

- Est une instance de la classe
- On peut créer plusieurs objets à partir d'une même classe

```
Rectangle R1;
Rectangle R2;
```


Principe (1)

- Les détails d'implémentation n'ont pas à être connus par l'utilisateur

Exemple : On crée un objet de type rectangle

```
int main() {
	Rectangle R1; // R1 est un objet de type Rectangle
	int a = R1.calculer_Perimetre();
	return 0;
}
```

- La fonction Calculer_perimetre() renvoie le périmètre
- Les détails de cette fonction se trouve dans la classe rectangle

Principe (2)

- Pour pouvoir créer des objets de type Rectangle, il va falloir coder la classe Rectangle

Exemple : On crée la classe rectangle

```
class Rectangle {
public:
   double longueur;
   double largeur;
   double calculer_Perimetre();
};
```

- La classe Rectangle contient uniquement le corps de la classe

Principe (3)

- Maintenant, il reste à implémenter la fonction Calculer_perimetre();

Exemple : On implémente la fonction

```
double Rectangle::calculer_Perimetre(){
	return 2 * longueur + 2 * largeur;
}
```

- Dans la fonction calculer_Perimetre(), on peut utiliser toutes les variables propres à la classe Rectangle
- L'implémentation de la fonction se fait à l'extérieur de la classe

Exemple basique d'une classe

```
class Rectangle {
public:
   double longueur;
   double largeur;
   double calculer_Perimetre();
};
double Rectangle::calculer_Perimetre(){
      return 2 * longueur + 2 * largeur;
}
int main() {
      Rectangle R1; // R1 est un objet de type Rectangle
      return 0;
}
```


Champs et méthodes (1)

Définition : **Champ**

- Est une variable associée à l'objet
- Aussi appelé attribut

Définition : **Méthode**

- Est une fonction membre de l'objet
- Comme une fonction classique, elle peut contenir des paramètres

```
Rectangle R1;
R1.longeur = 40;          // longueur est un champ
R1.calculer_Perimetre();  // calculer_Perimetre() est une méthode
```


Champs et méthodes (2)

- Les méthodes sont déclarées dans une classe
- Le nom complet de la méthode Test de la classe T est T::Test
- Les méthodes peuvent accéder aux champs des objets de la classe. Ce sont les seules à pouvoir le faire (ou presque) !
- L'objet sur lequel elle sont appelées, n'apparait pas explicitement dans la définition des méthodes
- Par défaut, les paramètres apparaissant dans la définition d'une méthode sont ceux avec lesquelles la méthode sera appelée

Déclaration des champs et méthodes

- Les champs et les méthodes sont déclarés à l'intérieur de la classe
- Les méthodes peuvent être codées à l'intérieur ou à l'extérieur de la classe

Exemple : 2 façons d'implémenter la méthode `calculer_Perimetre()`

```
class Rectangle {
public:
   double longueur;
   double largeur;
   double calculer_Perimetre();
};

double Rectangle::calculer_Perimetre(){
   return 2 * longueur + 2 * largeur;
} // Codée à l'extérieur
```

```
class Rectangle {
public:
   double longueur;
   double largeur;
   double calculer_Perimetre(){
   return 2 * longueur + 2 * largeur;
   } // Codée à l'intérieur
};
```

Accès aux champs et méthodes

- Pour accéder aux attributs et méthodes, on utilise la notation :
  - « . » pour les objets
  - « -> » pour les pointeurs

Exemple :

```
int main() {
   Rectangle R1; // R1 est une instance statique
   Rectangle *R2 = new Rectangle; // R2 est une instance dynamique
   R1.calculer_Perimetre();
   R2->calculer_Perimetre();
   return 0;
}
```


Visibilité des champs et méthodes (1)

- La visibilité des champs et des méthodes est définie dans l'interface de la classe. Trois mots-clés permettent de la préciser :
  - Public : autorise l'accès pour tous
  - Private : restreint l'accès aux méthodes de cette classe
  - Protected : comme private, restreint l'accès aux méthodes de cette classe, sauf que l'accès est aussi autorisé aux méthodes des classes qui héritent (directement ou indirectement) de cette classe.

Visibilité des champs et méthodes (2)

Exemple :

```
class Personne {
public:
   string nom;
   string prenom;
private:
   int age;
protected:
   string adr;
};
```

```
int main() {
   Personne P1; // Déclaration d'une personne P1
   cout << "Le nom de P1 : " << P1.nom << endl;
   cout << "L'age de P1 : " << P1.age << endl;
   cout << "L'adresse de P1 : " << P1.adr << endl;
   return 0;
}
```

- L'affichage de l'âge donne une erreur car le champ age est private
- L'affichage de l'adresse donne une erreur car le champ adr est protected

Constructeurs (1)

Définition : **Constructeur**

- Méthode particulière appelée automatiquement à chaque création d'un objet

Principe :

- Pour appeler un constructeur de la classe MaClasse, il suffit de faire suivre le nom de l'objet O par la liste des arguments

  MaClasse O (arg1, arg2, …);

- Lorsqu'aucun argument n'est donné, le constructeur sans argument est appelé

  MaClasse O;

Constructeurs (2)

Exemple : Un constructeur qui initialise la longueur et la largeur du rectangle

```
class Rectangle {
   public:
   Rectangle(double init_longeur, double init_largeur);
   double longueur = 0;
   double largeur = 0;
};

Rectangle::Rectangle(double init_longeur, double init_largeur) {
   longueur = init_longeur;
   largeur = init_largeur;
}
```


Constructeurs (3)

Exemple : Un constructeur qui initialise la longueur et la largeur du rectangle

```
int main() {
   Rectangle R1; // Appel au constructeur par default
   Rectangle R2(10, 20); // Appel au constructeur Rectangle(double, double)
   cout << "Longueur : " << R1.longueur << "Largeur : " << R1.largeur << endl;
   cout << "Longueur : " << R2.longueur << "Largeur : " << R2.largeur << endl;
   return 0;
}
```

- La longueur et la largeur affichées de R1 sont (0, 0)
- La longueur et la largeur affichées de R2 sont (10, 20)

Destructeurs (1)

Définition : **Destructeur**

- Méthode particulière appelée automatiquement à la destruction d'un objet

Principe :

- Son nom est de la forme :

  ~MaClasse()

- Par défaut, le destructeur ne fait rien. Toutefois, on peut lui donner un comportement spécifique
- Il est indispensable lorsque l'on a besoin de faire de l'allocation dynamique (quand il y'a des pointeurs dans la classe)

Destructeurs (2)

Exemple : Déclaration d'un destructeur

```
class Rectangle {
   public:
   ~Rectangle();
   double longueur = 0;
   double largeur = 0;
   MaStructure * ptr;
};

Rectangle::~Rectangle() {
   delete [] ptr; // Objectif du destructeur : libérer tous les pointeurs
}
```

Constructeurs par copie (1)

Définition : **Constructeur par copie**

- Méthode particulière appelée lors de l'instanciation d'un objet avec en argument un objet du même type

Principe :

- Son nom est de la forme :

  ~MaClasse(Const MaClasse &s)

- Le rôle d'un constructeur par copie est de permettre l'instanciation d'un nouvel objet dans le même état qu'un objet existant (clonage)
- Le constructeur copieur par défaut fait une initialisation champ à champ
- Comme pour le destructeur, il est utile d'implémenter un constructeur copieur uniquement dans les allocations dynamiques

Constructeurs par copie (2)

Exemple : Déclaration d'un constructeur par copie

```
class Rectangle {
   public:
   Rectangle(const Rectangle &s);
   double longueur = 0, largeur = 0;
   MaStructure * ptr;
};

Rectangle::Rectangle(const Rectangle &s) {
   longueur = s.longueur; largeur = s.largeur;
   ptr = new MaStructure[100];
   for (int i=0; i<100; i++)
      ptr[i] = s.ptr[i];
};
```


Constructeurs par copie (3)

Exemple : Un constructeur par copie

```
int main() {
   Rectangle R1;
   Rectangle R2(R1); // Appel au constructeur par copie
   Rectangle R3=R1; // Appel au constructeur par copie
   cout << "Longueur : " << R2.longueur << "Largeur : " << R3.largeur << endl;
   fonction(R1);
   return 0;
}
void fonction(Rectangle R4) {…} // Appel au constructeur par copie
```

- La longueur et la largeur affichées sont celles de R1

Exemple : Tableau d'étudiants

- Soit une classe Tab_etud qui permet de gérer un tableau de structures de type `Etudiant`
  - Chaque structure `Etudiant` contient le nom et la moyenne d'un étudiant
- Cette classe contient :
  - 2 champs : taille et un pointeur `ptr` qui pointe sur la structure `Etudiant`
  - 5 méthodes : element, supprimer, affiche, existe et ajout
  - 3 méthodes spéciales : constructeur, constructeur copieur et destructeur

Remarque : `ptr` joue le rôle d'un tableau de type Etudiant : `ptr[0]` pointe vers le premier étudiant `ptr[1]` le second etc …

La classe Tab_etud

```
class Tab_etud {
private:
	Etudiant * ptr;

protected:
	int taille;

public:
	Tab_etud(int);
	Tab_etud(const Tab_etud &s);
	~Tab_etud();

	bool existe(char * );
	void ajout(Etudiant ) ;
	Etudiant element(int );
	void supprimer(int );
	void affiche();
	};
```

```
struct Etudiant {
	char nom[20];
	float moyenne;
};
```


Les méthodes de Tab_etud

```
Etudiant Tab_etud::element(int position) {
	Etudiant e = ptr[position];
	return e;
}

void Tab_etud::supprimer(int position) {
for(int i = position; i < taille - 1; i++)
	{ptr[i] = ptr[i+1];}
taille = taille -1;
}

void Tab_etud::affiche() {
cout << " Voici la liste :\n";
for (int i = 0; i < taille ; i++)
	cout << " Nom : " << ptr[i].nom <<
	 " Moyenne : " << ptr[i].moyenne ;
}
```

```
bool Tab_etud::existe(char * nom) {
for(int i =0; i< taille; i++)
	{//strcmp renvoie 0 si identique
	if( strcmp(ptr[i].nom, nom) == 0)
		return true;
	}
return false;
}

void Tab_etud::ajout(Etudiant e) {
if( !existe(e.nom) ) {
	// Ajout dans la dernière position
	ptr[taille] = e;
	taille = taille + 1;
	}
}
```


Les méthodes spéciales de Tab_etud

```
Tab_etud::Tab_etud(int n) {
	ptr = new Etudiant[n];
	taille = 0;
}

Tab_etud::Tab_etud(const Tab_etud &s) {
	taille = s.taille;
	ptr = new Etudiant[100];
	for(int i=0;i<taille;i++)
		{ptr[i] = s.ptr[i];}
}

Tab_etud::~Tab_etud() {
	// suppression du pointeur
	delete [] ptr;
}
```


Fonctions amies

Définition : **Fonctions amies**

- Si une fonction F est amie « friend » d'une classe C1, alors F peut accéder aux champs privés de C1.
- Si une classe C2 est amie de C1, alors toutes les fonctions membres de C2 peuvent accéder aux champs privés de C1.

Exemple :

```
class C1 {
	friend void F() ;
	friend class C2 ;
	…
};
```

Surcharge d'opérateurs (1)

- Il est possible de surcharger les operateurs (+, -, [ ], =, ==, ...) de C++
- Il existe 2 façons de faire :

Fonction globale : A op B qui est vue comme l'application d'une fonction op à deux arguments A et B

MaClasse operator + (MaClasse A, MaClasse B)

Fonction membre : A op B qui est vue comme l'application d'une fonction op à un argument B de l'objet A

MaClasse MaClasse :: operator + (MaClasse B)

- Réalisons la surcharge de (+) de la classe Tab_op (identique à Tab_etudiants)

Surcharge d'opérateurs (2)

- Soit une classe Tab_op qui dérive de la classe Tab_etud (voir Héritage)
  - Cette classe contient les mêmes champs et méthodes que Tab_etud
- Dans cette classe, on définit :
  - Un opérateur + permettant de concaténer deux tableaux. Ça consiste à mettre les tableaux, l'un à la suite de l'autre
  - Un opérateur = permettant de copier un tableau dans un autre

Objectif : Implémenter les opérateurs + et = et réaliser des opérations du genre :

```
Tab_op A(100), B(100);
Tab_op C = A + B; // Utilisation du constructeur de l'objet C
Tab_op D(100);
D = A + B; // Utilisation de l'opérateur = de l'objet D
```


Surcharge d'opérateurs (3)

- Par fonction globale :

```
//  Opérateur + :
Tab_op operator + ( Tab_op a, Tab_op b ) {
      // Principe : on retourne a + b
      Tab_op result(a); // Copie de a
      for( int i = 0; i < b.taille; i++ )
            {result.ajout(b.element(i));} // Ajout des éléments de b
      return result;
}
```

Remarque : Dans les fonctions globales on prend 2 paramètres. Les fonctions globales sont définies amies et accèdent aux champs privés de a et b

Surcharge d'opérateurs (4)

- Par fonction membre :

```
//  Opérateur + :
Tab_op Tab_op::operator + ( Tab_op  b ) {
      // Principe : on retourne *this + b
      Tab_op result(* this); // Copie this
      for( int i = 0; i < b.taille; i++ )
            {result.ajout(b.element(i));}
            // Ajout des éléments de b
      return result;
}
```

```
//  Opérateur = :
void Tab_op::operator = (Tab_op b)
{
taille = 0;
for (int i = 0; i < b.taille; i++)
      this->ajout(b.element(i));
      // Ajout des éléments de b
}
```

Remarque : Dans les fonctions membres a = this. Le retour de a + b représente le résultat. Le retour de a = b n'est pas important, il faut toutefois mettre b dans a

Cours 3 : L'héritage

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Principaux axes du cours

- La notion d'héritage
- Les types d'héritage
- Surcharge de méthodes
- Constructeurs dans l'héritage
- Héritage multiple
- Héritage virtuel

Principe

Définition : **L'héritage**

- Aussi appelé dérivation
- L'héritage permet de définir une nouvelle classe en se basant sur une classe déjà existante qu'on appelle la « classe mère »

Avantages :

- Réutilisation totale ou partielle des classes déjà développées
- Gain de temps et d'énergie pour le développement et la maintenance des codes sources
- Les méthodes identiques ne sont codées qu'une fois dans la classe mère

Implémentation (1)

Syntaxe :

- On utilise l'opérateur : et on spécifie le type d'héritage
- La classe B hérite publiquement de la classe A

```
class B : public A
```

Exemple : La classe Carre

- Les attributs : longueur et largeur et la méthode : calculer_Perimetre() ne sont déclarés qu'une fois dans la classe mère Rectangle
- La classe Carre hérite de la classe Rectangle

```
class Carre : public Rectangle
```

Implémentation (2)

```
class Rectangle {
public:
	Rectangle(int lon, int lar);
	int longueur;
	int largeur;
	int calculer_Perimetre();
};

class Carre {
public:
	Carre(int cote);
	int longueur;
	int largeur;
	int calculer_Perimetre();
};

int main() {
	Rectangle R1(10, 10);
	Carre R2(10);
	cout << R1.calculer_Perimetre();
	cout << R2.calculer_Perimetre();
	return 0;
}
```

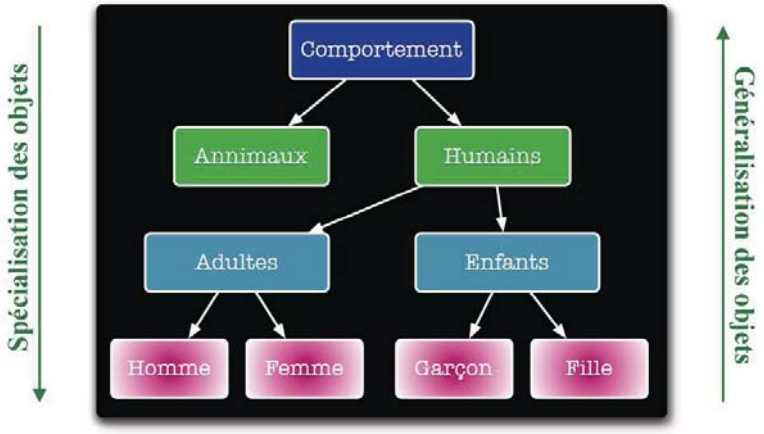

Implémentation (3)

```
class Rectangle {
public:
	Rectangle(int lon, int lar);
	int longueur;
	int largeur;
	int calculer_Perimetre();
};

class Carre : public Rectangle {
public:
	Carre(int cote);
};

int main() {
	Rectangle R1(10, 10);
	Carre R2(10);
	cout << R1.calculer_Perimetre();
	cout << R2.calculer_Perimetre();
	return 0;
}
```

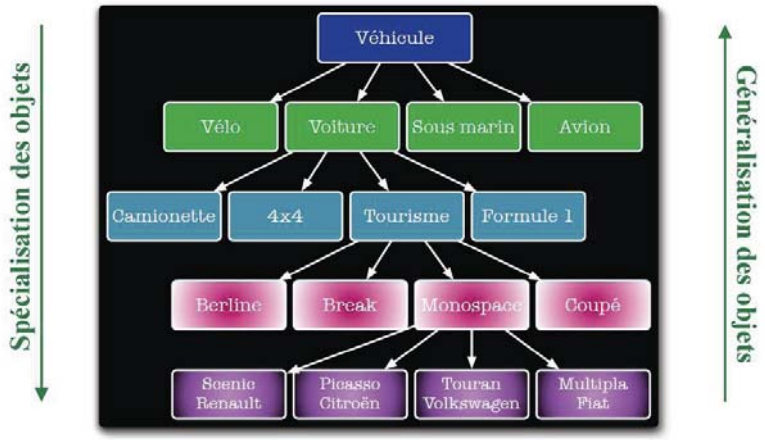

Propriétés

- L'héritage permet de donner à une classe toutes les caractéristiques d'une autre classe ou de plusieurs autres classes (héritage multiple)
- La classe qui hérite est appelée classe fille, classe dérivée ou classe descendante
- Les classes dont elle hérite sont appelées classes mères, classes de base ou classes antécédentes
- Les propriétés héritées par la classe fille sont : les attributs et les méthodes
- Après héritage, les classes filles peuvent définir de nouveaux membres ou redéfinir des méthodes de la classe mère

Exemple (1)



Exemple (2)



Exemple (3)

- Exemple 1 : La classe Garçon

```
class Garçon : public Enfants { // … // };
class Enfants : public Humains { // … // };
class Humains : public Comportements { // … // };
```

- Exemple 2 : La classe BMW320D

```
class BMW320D : public Berline { // … // };
class Berline : public Tourisme { // … // };
class Tourisme : public Voiture { // … // };
class Voiture : public Véhicule { // … // };
```

- Exemple 3 : La classe Tab_op

```
class Tab_op : public Tab_etud { // … // };
```


Types d'héritage

- Il existe 3 types d'héritage :
  - Public
  - Protected
  - Private
- Le mode d'héritage détermine quels sont les méthodes et les attributs de la classe de base qui seront accessibles dans la classe dérivée
- Si aucun mode d'héritage n'est spécifié, C++ prend par défaut : private
- Les membres privés de la classe de base ne sont jamais accessibles par les membres des classes dérivées

L'héritage publique

Définition : **L'héritage publique**

- Donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée
- C'est la forme la plus courante d'héritage, car il permet de modéliser : "B est une sorte de A" ou "B est une spécialisation de A"

Syntaxe :

- La classe B hérite de façon publique de la classe A

```
class B : public A
```

L'héritage privé

Définition : **L'héritage privé**

- Donne aux membres publics et protégés de la classe de base le statut de membres privés dans la classe dérivée
- Très peu utilisé (car pas utile), il permet de modéliser : "B est composé de A"

Syntaxe :

- La classe B hérite de façon privée de la classe A

```
class B : private A
```


L'héritage protégé

Définition : **L'héritage protégé**

- Donne aux membres publics et protégés de la classe de base le statut de membres protégés dans la classe dérivée
- Il est utilisé lorsque l'on souhaite que des méthodes soient accessibles aux futures dérivées de la classe

Syntaxe :

- La classe B hérite de façon protégée de la classe A

```
class B : protected A
```


Surcharge de méthodes (1)

Définition : **Surcharge de méthodes**

- Correspond à redéfinir une fonction dans une classe dérivée
- Il est nécessaire que la méthode possède le même nom et les mêmes attributs que dans la classe de base

Principe :

- Il est possible de choisir quelle méthode on désire exécuter
  - La méthode surchargée
  - La méthode de la classe de base (avec l'opérateur ::)

Surcharge de méthodes (2)

```
class A {
public:
	int getValue() { return 0; }
};

class B : public A {
public:
	int getValue() { return 1; }
};
```

```
int main() {
	B b;
	cout << b.getValue(); // -> 1
	cout << b.A::getValue(); // -> 0
	return 0;
}
```

Constructeurs dans l'héritage (1)

- Les constructeurs, constructeur de copie, destructeurs et opérateurs d'affectation ne sont jamais hérités
- Les constructeurs par défaut des classes de bases sont automatiquement appelés avant le constructeur de la classe dérivée
- L'appel des destructeurs se fait dans l'ordre inverse des constructeurs
- Remarque : Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, on utilise une liste d'initialisation

Constructeurs dans l'héritage (2)

- Constructeurs par défaut et constructeurs avec une liste d'initialisation

```
class A {
public: A() {}      // Constructeur par default
       A(int n) {}  // Autre constructeur
};

class B : public A {
public: B() {}      // Constructeur par default
B(int i) : A(i) {}  // Liste d'initialisation
};

int main() {
       B b1;        // A() -> B()
       B b2(7);     // A(7) -> B(7)
       return 0;
} // Ordre d'appel des constructeurs : A -> B
```

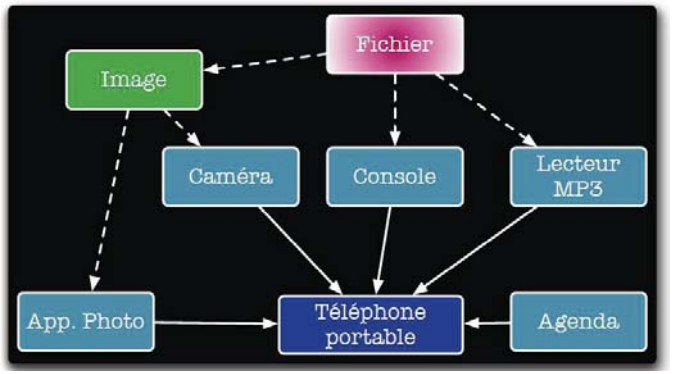
```
class Tab_op : public Tab_etud {
public:
Tab_op(int n) : Tab_etud(n) {}
//Liste d'initialisation
void supprimer();
friend Tab_op operator + (Tab_op,
Tab_op);
//Tab_op operator + (Tab_op) ;
void operator = (Tab_op);
};
```


Constructeurs dans l'héritage (3)

```
class Animal {
public:
       Animal() {cout << "Animal" << endl;}
       ~Animal() {cout << "~Animal" << endl;}
};

class Chien : public Animal {
public:
       Chien() {cout << "Chien" << endl;}
       ~Chien() {cout << "~Chien" << endl;}
};
```

```
int main() {
       Chien *york = new Chien();
       delete york;
       return 0;
}

       /* Le programme affiche :
       Animal
       Chien
       ~Chien
       ~Animal
       */
```


Héritage multiple (1)



Héritage multiple (2)

- Le langage C++ permet de réaliser des héritages multiples
- Pour chaque classe de base, on peut définir le mode d'héritage : public, private ou protected
- L'héritage multiple peut mener à une complexification du code source et de sa compréhension

Syntaxe :

class C : public B, public A

Héritage multiple (3)

```
class A {
public:
      int dataA; // l'attribut dataA de la classe A
};
class B {
public:
      int dataB; // l'attribut dataB de la classe B
};
class C : public B, public A {
public:                                          // Dans la classe C, nous utilisons
      int somme() { return dataA + dataB; }      // les attributs des classes A et B
};                                               // dont nous héritons
```


Constructeurs dans l'héritage multiple

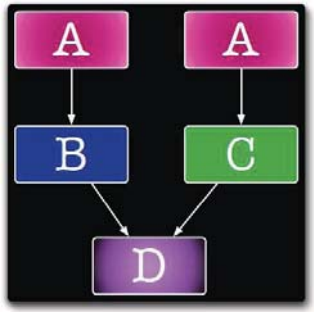
- Les constructeurs sont appelés dans l'ordre de déclaration de l'héritage
- Les destructeurs sont appelés dans l'ordre inverse de celui des constructeurs

```
class A {
public: A(int n = 0) { /* ... */ }
};
class B {
public: B(int n = 0) { /* ... */ }
};
class C : public B, public A {
public: C(int i, int j) : A(i), B(j)
{ /*...*/ }
};
```

```
int main() {
      C c1; // B() -> A() -> C()
      C c2(5,4); // B(4) -> A(5) -> C(5,4)
      return 0;
}
      // ordre d'appel des constructeurs
      // B -> A -> C
```


Héritage virtuel (1)

- Problème : (héritage multiple) Un objet de la classe D contiendra 2 fois les données héritées de A
  - Une fois par héritage de la classe B
  - Une autre fois par héritage de C



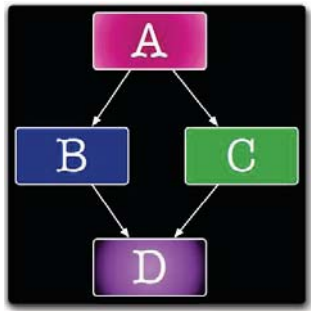
Supposons que A contienne une variable var :

```
int main() {
      D obj;
      obj.var = 0; // Ambiguité (erreur)
      obj.B::var = 1; // OK
      obj.C::var = 2; // OK
      return 0;
}
```

Héritage virtuel (2)

- Solution : On souhaite une unique occurrence des membres de la classe mère

Pour que la classe D n'hérite qu'une seule fois de la classe A, il faut que les classes B et C héritent virtuellement de A




```
class A { int var; }
class B : virtual public A { /* … */ };
class C : virtual public A { /* … */ };
class D : public B : public C { /* … */ };
```


Cours 4 : Le polymorphisme

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Principaux axes du cours

- La notion de polymorphisme
- Polymorphisme des fonctions
- Polymorphisme dans l'héritage
- Compatibilité des objets et conversion
- Mot clé virtual : héritage virtuel, méthodes virtuelles et méthodes virtuelles pures
- Abstraction dans l'héritage

Principe (1)

Définition : **Polymorphisme**

- = Plusieurs formes
- Pour les méthodes
- Pour les objets dans l'héritage

On parle alors de polymorphisme dans plusieurs contextes différents :

- Polymorphisme des fonctions
- Polymorphisme dans l'héritage

Principe (2)

Polymorphisme des fonctions :

- La possibilité de définir plusieurs fonctions avec : le même nom et des paramètres de type et de nombre différent

Polymorphisme dans l'héritage :

- La capacité d'appeler une méthode en fonction du type de l'objet appelant (sa hiérarchie dans l'héritage)
- La possibilité de traiter plusieurs formes d'une classe :
  - Le cas de plusieurs classes héritant d'une classe abstraite

Polymorphisme des fonctions (1)

- Contrairement au langage C, le langage C++ intègre le polymorphisme des fonctions :

Exemple : `fct(int a), fct(int a, int b), fct(int a, string b) …`

- Chaque fonction est identifiée par une « clef » basée sur son nom et le type et l'ordre des paramètres qu'elle accepte

Remarque :

Le polymorphisme de fonctions peut aussi désigner le fait de pouvoir définir des fonctions de même nom et paramètres dans des classes différentes

Polymorphisme des fonctions (2)

```
class Nombre {
public:
      int valeur;
      Nombre() { } // Const par défault
      Nombre(int val){ valeur = val;}
      void affiche(int val)
            { cout << val << endl; }
      void affiche(string val)
            { cout << val << endl; }
      void affiche(Nombre *objet)
            { cout << objet->valeur <<
            endl; }
};
```

```
#include <string>
int main() {
      Nombre A;
      Nombre * B = new Nombre(7);
      A.affiche("3");
      A.affiche(5);
      A.affiche(B);
      return 0;
}
// Le programme affiche : 3, 5 et 7
```


Polymorphisme dans l'héritage

Compatibilité des objets :

- Tout objet d'une classe dérivée peut être traité et utilisé comme un objet de sa classe de base

Méthodes virtuelles :

- Méthodes virtuelles pour gérer la redéfinition des fonctions (surcharge)
- Méthodes virtuelles pures pour créer des classes abstraites

Compatibilité des objets et conversion (1)

- Une conversion implicite d'un objet d'une classe dérivée B en un objet de la classe de base A est possible si l'héritage est public
- L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres supplémentaires de la classe dérivée

Principe de conversion :

Classe fille ➔ Classe mère Possible
Classe mère ➔ Classe fille Impossible

Compatibilité des objets et conversion (2)

Exemple :

```
class A {};
class B : public A {};
int main() {
      A a;
      B b;   // B est une classe dérivée de A
      a = b; // correct, seule la partie A de b est copiée dans a
      b = a; // incorrect, impossible de mettre A dans B. Il manque des champs
      A* pa = &a;  // correct bien sur
      pa = &b;     // correct aussi
      B* pb = &a;  // erreur
      return 0;
}
```


Mot clé virtual

Héritage multiple : ➔ Permet de ne pas dupliquer les attributs et méthodes de A dans D

- `class B : virtual public A {};`
- `class C : virtual public A {};`
- `class D : public B : public C {};`

Méthodes virtuelles : ➔ Permet d'appeler la méthode `fct ()` en fonction de la nature de l'objet

- `virtual void fct ();`

Méthodes virtuelles pures : ➔ Permet de créer une classe abstraite

- `virtual void fct () =0;`

Méthodes virtuelles (1)

- Le polymorphisme consiste en les différentes formes que peut prendre l'implémentation d'une méthode définie plusieurs fois (surchargée)
- Cette méthode change en fonction du type d'objet qui l'appelle
- En C++, on parle alors de méthode virtuelle : méthode à laquelle on rajoute le mot-clé virtual dans la définition

Syntaxe : `virtual void fct ();`

- Considérons l'exemple suivant qui se base sur les Carres et Rectangles

Méthodes virtuelles (2)

```
class Rectangle {
public: Rectangle(int lon, int lar);
        int longueur;
        int largeur;
        int calculer_perimetre()
        {return 2 * longueur + 2 *
        largeur; }
};
class Carre : public Rectangle {
public: Carre(int cote);
        int calculer_perimetre()
        {return 4 * largeur; }
};
```

```
int main() {
        Rectangle R1(10, 10);
        Carre R2(10);
        Rectangle * ptr;
        ptr = &R2;
        cout << ptr->calculer_Perimetre()
        << endl;
}
// Ici on a une liaison statique, le
// compilateur utilise la fonction
// calculer_Perimetre() de Rectangle
```


Méthodes virtuelles (3)

- Dans le cas d'une liaison statique (décidée par le compilateur) la seule fonction qui sera appelée est `Rectangle::calculer_Perimetre()`
- Ce cas a peu d'intérêt, car nous avons défini une fonction `Carre::calculer_Perimetre()`

Problème :

On aurait souhaité exécuter `Carre::calculer_Perimetre()` dans le cas où `ptr` pointe sur une instance de type Carre

Méthodes virtuelles (4)

Solution :

Rendre la fonction `calculer_Perimetre()` virtuelle dans les classes Rectangle et Carre

- En demandant que la fonction `calculer_Perimetre()` soit définie virtuelle, on impose une liaison dynamique
- La fonction à appeler sera alors déterminée à l'exécution en fonction du type de l'objet

Méthodes virtuelles (5)

```
class Rectangle {
public: Rectangle(int lon, int lar);
        int longueur;
        int largeur;
        virtual int calculer_perimetre()
        {return 2 * longueur + 2 *
        largeur; }
};
class Carre : public Rectangle {
public: Carre(int cote);
        virtual int calculer_perimetre()
        {return 4 * largeur; }
};
```

```
int main() {
        Rectangle R1(10, 10);
        Carre R2(10);
        Rectangle * ptr;
        ptr = &R2;
        cout << ptr->calculer_Perimetre()
        << endl;
}
// Ici on a une liaison dynamique, le
// compilateur utilise la fonction
// calculer_Perimetre() de Carre
```

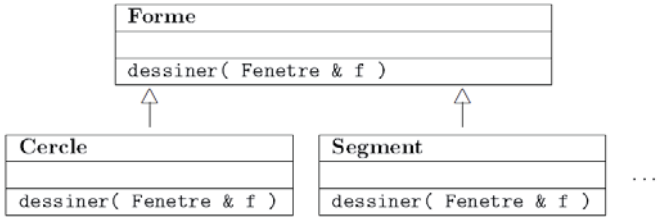
Méthodes virtuelles pures (1)

- Une méthode virtuelle pure est une méthode qui est déclarée mais non définie dans une classe
- Elle est définie dans une des classes dérivées de cette classe
- Pour déclarer une méthode virtuelle pure dans une classe, il suffit de faire suivre sa déclaration de « =0 » :
- Syntaxe : `virtual void fct () =0;`
- L'information « =0 » signifie ici simplement qu'il n'y a pas d'instance de cette méthode dans cette classe

Méthodes virtuelles pures (2)

Exemple : Un programme manipule différentes formes géométriques :

- Segment de droite
- Carre
- Rectangle
- Cercle
- ...



Objectif : On souhaite surcharger la méthode dessiner dans l'ensemble des formes géométrique

Méthodes virtuelles pures (3)

```
class Forme {
      virtual void dessiner(Fenetre & f) = 0;
      // Notez le '= 0' transforme la classe 'Forme' en classe abstraite
};
class Segment : public Forme {
      virtual void dessiner(Fenetre & f) {/*…*/}
      // La classe Segment contient sa propre définition de dessiner
};
class Cercle : public Forme {
      virtual void dessiner(Fenetre & f) {/*…*/}
      // La classe Cercle contient sa propre définition de dessiner
};
```


Abstraction dans l'héritage (1)

Définition : Une classe est dite abstraite si elle contient au moins une méthode virtuelle pure

- On ne peut pas créer d'instance d'une classe abstraite
- Une classe abstraite ne peut pas être utilisée comme argument ou type de retour d'une fonction

Objectif :

- Les méthodes virtuelles pures servent de cadre générique pour les méthodes virtuelles des classes dérivées
- Ceci permet de garantir une bonne homogénéité de l'architecture des classes

Abstraction dans l'héritage (2)

Remarque :

Si une classe hérite d'une classe abstraite, mais ne définit pas de corps à une méthode virtuelle pure héritée :

➔ Cette nouvelle classe est toujours une classe abstraite, non instanciable

Exemple :

```
class A {
public:
      virtual void fct() = 0;
};
class B : public A {};
```

```
int main() {
      A a;   // incorrect, A abstraite
      B b;   // incorrect, B abstraite
}            // tant que fct() non défini
```


Cours 5 : Les interfaces graphiques

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Principaux axes du cours

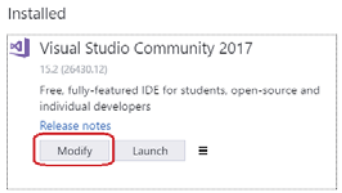
- Introduction au CLR / CLI Support
- Installer VS 2017 avec CLR / CLI Support
- Mettre en place un projet avec CLR / CLI Support

CLR / CLI Support

- Common Language Runtime (**CLR**) est le nom choisi par Microsoft pour la machine virtuelle du framework **.NET**
- Common Language Infrastructure (**CLI**) définit l'environnement d'exécution des codes de programmes du framework **.NET**
- Plusieurs langages reposent sur cette technologie puissante du .NET tel que : C# et ASP
- Nous utiliserons CLR / CLI Support pour créer des interfaces graphiques

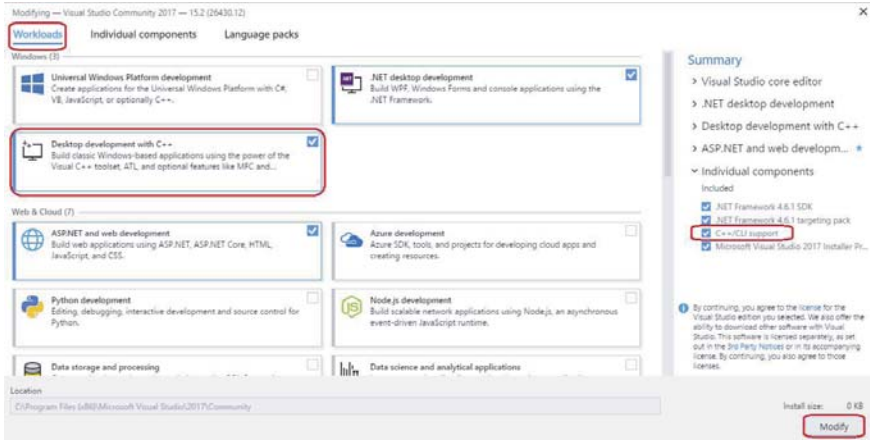
Installation (1)

- Assurez vous d'avoir installer Visual Studio 2017 avec C++ et le package CLR / CLI Support

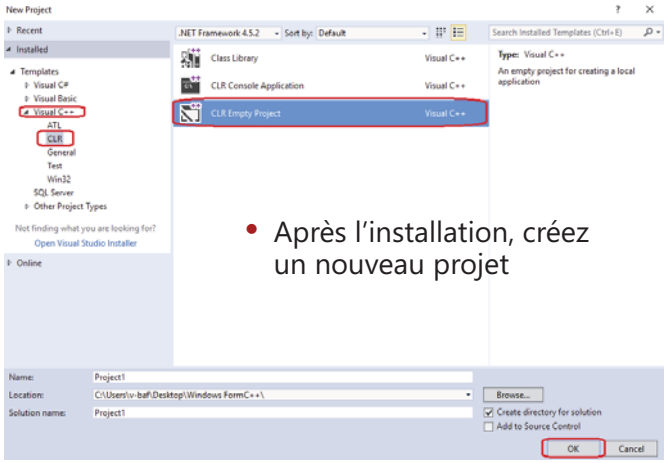


Installation (2)

Installation (3)

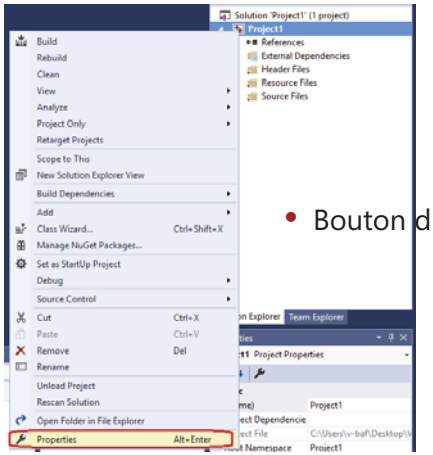


Mise en place d'un projet (1)



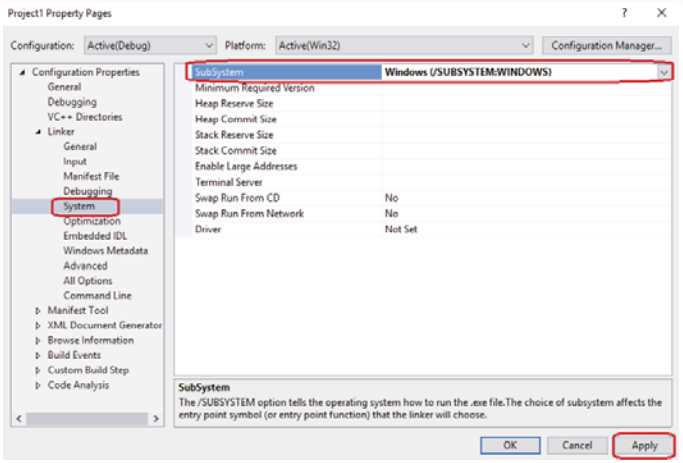
- Après l'installation, créez un nouveau projet

Mise en place d'un projet (2)

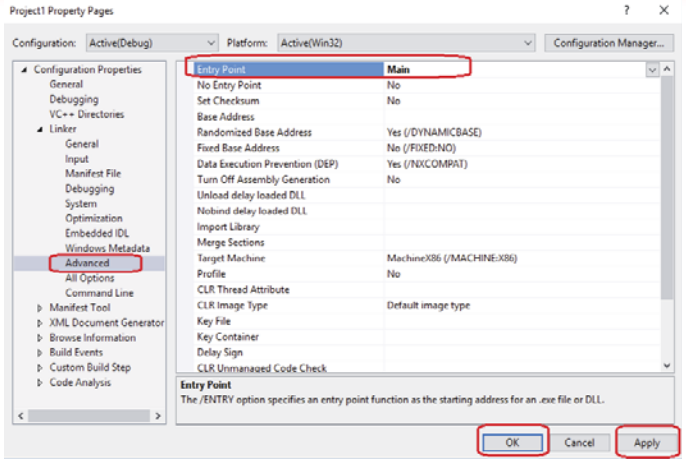


- Bouton droit sur Project1

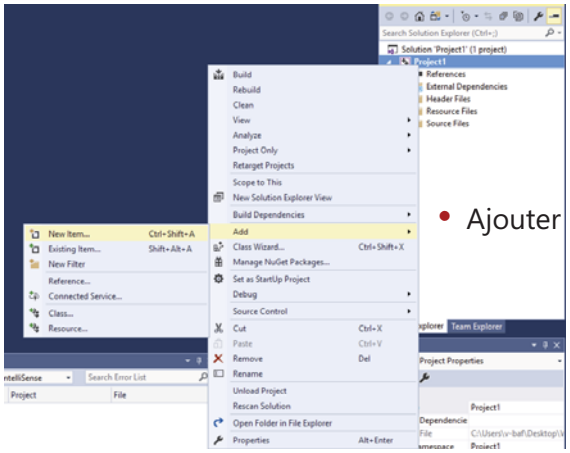
Mise en place d'un projet (3)



Mise en place d'un projet (4)

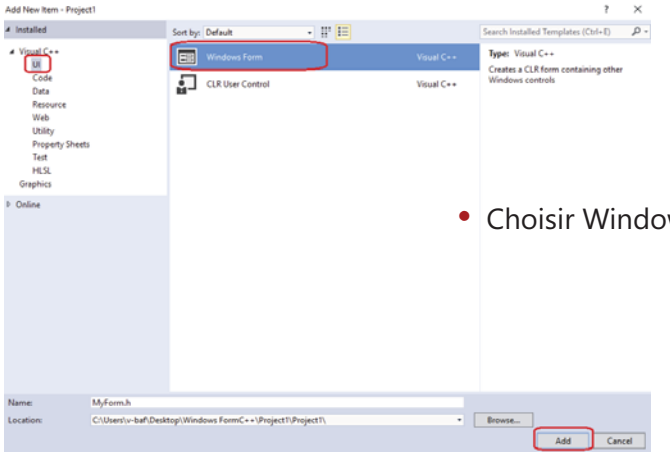


Mise en place d'un projet (5)



- Ajouter un nouvel élément

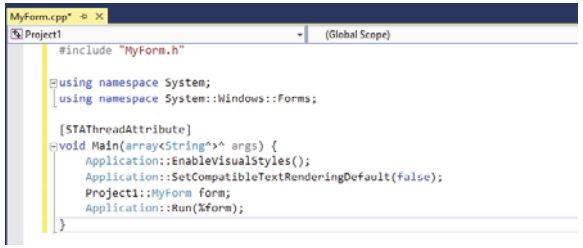
Mise en place d'un projet (6)



- Choisir Windows Form

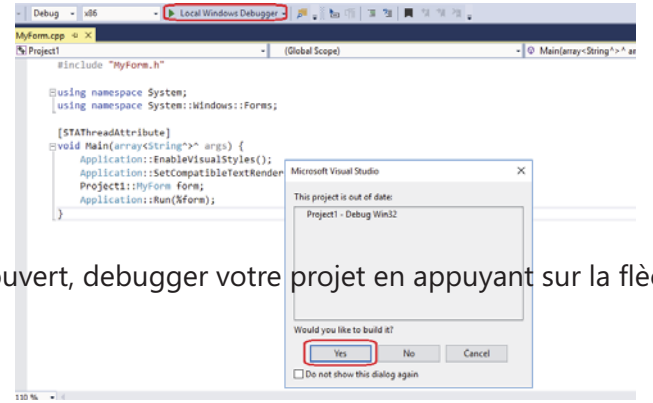
Mise en place d'un projet (7)

- Fermez cette fenêtre d'erreur, et copiez le code suivant à MyForm.cpp:



Mise en place d'un projet (8)

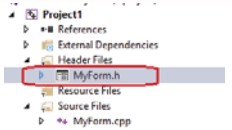
- Enregistrez, fermez et réouvrez VS 2017, puis votre projet1



- Une fois ouvert, debugger votre projet en appuyant sur la flèche verte

Mise en place d'un projet (9)

- En cliquant sur MyForm.h



- Vous avez accès à la Toolbox : button, label, picturebox, etc …