

Programmation objet 2 et outils de développement

Licence 3 Informatique

Année universitaire 2009-2010
David Genest

Chapitre I

Introduction

- 1 Présentation du cours
- 2 De C à C++

Chapitre I

Introduction

- 1 Présentation du cours
- 2 De C à C++

Programmation objet 2

Objectif du cours

- Approfondir les concepts de la programmation orientée objet. (Déjà vus en Java dans le cours de POO)
- Apprendre le langage C++.
- Utiliser des outils d'aide au développement. Environnement de développement, débogueur, documentation du code, profilage. . .

Organisation

- 12h de cours : Concepts de la POO et C++.
- 18h de TD, 25h de TP : Programmation C++ et outils de développement.

Évaluation

- $\frac{1}{3}$ CC + $\frac{2}{3}$ Examen.
- Contrôle continu : TP relevé et noté.

Introduction POO

Programmation Orientée Objet

Paradigme (style) de programmation consistant à assembler des *briques logicielles* (*objets*).

Un *objet* représente un *concept* ou une *entité* du monde.

Mais les objets ne sont pas uniquement utilisés pour la programmation. . .

- Bases de données objets (Oracle)
- Méthodes de conception objets (UML)
- HTML+CSS, PHP . . .

Objectif premier

Faciliter le développement d'une application robuste.

Introduction POO

Comparaison avec la programmation impérative (C, Pascal, . . .)

- **Impératif.** Un problème est décomposé en sous-problèmes, une fonction est écrite pour résoudre chaque problème. Les fonctions s'appellent entre elles en se passant des paramètres.
- **Objets.** On identifie les concepts (ou entités) du problème, ainsi que les relations entre ces concepts. Chacun de ces concepts est ensuite représenté par les données qu'il contient et par les traitements qu'il est capable d'effectuer.

Langage à classes

La plupart des langages de programmation OO sont des langages à classes : une classe définit un « moule » pour un ensemble d'objets qui ont la même structure et fournissent les mêmes traitements.

Introduction C++

C++

- Langage normalisé par l'ISO
- Défini dans les années 1980 (mais a évolué depuis)
- « Amélioration » de C
 - ⇒ facilite l'apprentissage pour quelqu'un qui connaît déjà C
 - Mais attention... faire du C en C++ n'est pas programmer en C++!
 - Réutilisation facilitée de code/bibliothèques C
- Doté d'une bibliothèque de classes et algorithmes
- Portable

Introduction C++

Objectifs

- Efficacité (mémoire et rapidité)
 - Le langage n'ajoute aucune fonction « cachée » demandant de la mémoire ou des traitements
 - Pas de ramasse-miettes (garbage collector)
- Développement plus rapide
 - Utilisation de la bibliothèque standard, Réutilisation du code
- Gestion de la mémoire plus simple
 - Gérer les libérations de mémoire (`free`) plus facilement,
 - Gestion des chaînes plus simple
- Vérification plus stricte des types
- Avantages de la POO
- ...

Chapitre I

Introduction

- 1 Présentation du cours
- 2 De C à C++
 - Types et variables
 - Allocation dynamique
 - const
 - Références
 - Conversion de type
 - Compilation

Types de données

- signed ; unsigned
- char ; short (short int) ; int ; long (long int)
; float ; double ; long double
- wchar_t (unicode)
- bool → true false

Exemple

```
int f(int a, int b)
{
    int res=0;
    for (int i=1; i<=a; i++)
        res += b;
    return res;
}
```

Portée des variables

Une variable est accessible à **partir de sa déclaration dans le bloc** (accolades) **dans lequel elle est déclarée.**

- Une variable globale est déclarée à l'extérieur de tout bloc. Il est toutefois fortement conseillé de ne pas utiliser de variables globales.
- Cas particulier d'une déclaration dans un `for` :
La variable est accessible uniquement dans les instructions répétées par le `for`.

Portée des variables

Exemple

```
#include <iostream>
using namespace std;
signed short varglobale;
int main()
{
    varglobale=5; // initialisation
    unsigned int varlocale=10;
    while (varglobale >= 0)
    { int v=6;
      varglobale += v;
      varlocale--;
    }
    cout << varglobale << endl;
    return 0;
}
```

Déclaration et initialisation des variables

Exemple

```
int a, b, c; // équivalent à  
int a; int b; int c;
```

Attention

```
int* a, b;
```

a est un pointeur sur un entier, b est un entier.

Exemple (Initialisation C / Initialisation C++)

```
int a=0; // équivalent à  
int a(0);
```

Chaînes de caractères

Le type (classe) `std::string` fournit les opérations courantes sur les chaînes de façon bien plus simple que les « chaînes » C.

- Déclaration et initialisation

```
string s1;  
string s2a("valinit"), s2b="valinit";  
string s3a(s1), s3b=s1;
```
- Affectation =
- Comparaison == != <= ...
- Accès à un caractère `s[i]`
- Lecture au clavier `std::cin >> s;`
- Affichage `std::cout << s;`
- Longueur d'une chaîne `s.length()`

Tableaux

Exemple

```
int t1[4];
int t2[4]={1,5,7,9};
int t3[]={12,42,23};
int t2d[4][2];
t2d[2][1] = 12;
t2d[4][2] = 23; // Attention !
```

Attention

Un tableau est passé **par pointeur** à une fonction.
Une fonction ne peut pas connaître la taille d'un tableau.

```
int maximum(int tab[], int taille) { ... }
cout << maximum(t1,4);
```

Nouvelle utilisation de typedef

typedef n'est plus utile lors de la déclaration d'une struct ou d'une enum.

Exemple

```
typedef unsigned int Annee;
enum EtatPersonne
{ MAJEUR,
  MINEUR
};
struct Personne
{ string nom; string prenom;
  EtatPersonne etat; Annee naissance;
};
...
Personne p;
p.nom="Durand";
```

Arguments par défaut

Les *derniers* arguments d'une fonction peuvent avoir une **valeur par défaut** lors de la déclaration de la fonction.

Si les arguments correspondants ne sont pas passés lors d'un appel, la valeur par défaut est prise en compte.

Exemple

```
void f(int a=2, int b=3)
{
    cout << a << " " << b << endl;
}
...
f();      // 2 3
f(5);    // 5 3
f(5,7);  // 5 7
```

Allocation dynamique en C

Syntaxe (en C)

- Allocation simple
`int * i = (int *) (malloc(sizeof(int)));`
- Allocation d'une zone
`int * t = (int *) (malloc(sizeof(int)*taille));`
- Libération
`free(t);`

Allocation dynamique

en C++

Syntaxe (en C++)

- Allocation simple

```
int * i = new int;  
int * i = new int(3);
```

 Fixer la valeur initiale
- Allocation d'une zone

```
int * t = new int[taille];
```
- Libération simple

```
delete i;
```
- Libération d'une zone

```
delete []i;
```

Allocation dynamique

en C++

Attention

- Tout ce qui est alloué doit être libéré.
- Tout ce qui est alloué par `new ... []` doit être libéré par `delete []`.
- Ne pas mélanger `new/delete` avec `malloc/free`.

const

Le mot-clef `const` spécifie que la valeur d'une variable est constante.

Exemple

```
const int i=4; // équivalent a  
int const i=4;
```

```
i = 5; // erreur de compilation
```

Une (variable) constante (mais pas une variable (non déclarée `const`)) peut être utilisée pour spécifier la taille d'un tableau. Souvent on marque syntaxiquement les constantes (écriture en majuscules, première lettre en majuscule, préfixe `c_ ...`)

const et pointeurs

Attention à la position du `const`

- Pointeur variable sur des caractères variables
`char * c;`
- Pointeur constant sur des caractères variables
`char * const c;`
`c = c1;` est interdit. `*c='a'`; `c[1]='b'`; sont autorisés.
- Pointeur variable sur des caractères constants
`const char * c;` (ou `char const * c;`)
`c = c1;` est autorisé. `*c='a'`; `c[1]='b'`; sont interdits.
- Pointeur constant sur des caractères constants
`const char * const c;` ou `(char const * const c;)`
`c = c1;` `*c='a'`; `c[1]='b'`; sont interdits.

Moyen mnémotechnique : lire à l'envers. « Un pointeur sur un caractère constant », « Un constant pointeur sur un caractère »...

Références

Une **référence** est une variable synonyme (qui porte le même nom) qu'une autre variable.

Syntaxe (Déclaration d'une référence)

```
type & nomvar=var_référencée ;
```

La variable référencée doit **obligatoirement** être précisée au moment de la *déclaration*.

Exemple

```
int i;  
int & j=i;  
  
i=5;  
cout << j << endl; // 5  
j=3;  
cout << i << endl; // 3
```

Références

Les références sont principalement utilisées pour le **passage par référence** d'un argument à une fonction. Dans ce cas, on ne doit **pas** préciser la variable référencée.

Exemple

```
void f(int & a, int & b)  
{ a=2; b=3; }  
...  
int c,d;  
f(c,d);  
cout << c << " " << d << endl; // 2 3
```

Simplifie dans la plupart des cas le « passage par pointeur » de C.

Conversion de type

Syntaxe (en C)

(nouveau type) expression

Exemple

```
int main()
{ int a=2, b=3;
  float f=a/b;
  cout << f << endl;
  return 0;
}
```

- Affiche 0
- Forcer la division sur les réels
float f=((float)a)/b;
- Affiche 0.666667

Conversion de type

Inconvénient de la syntaxe C :

Peu visible (syntaxe) alors que dangereux.

Syntaxe (en C++)

- `static_cast<nouveau type>(expression)`
Force la conversion de type de l'expression dans le nouveau type
float f=static_cast<float>(a)/b;
- `reinterpret_cast<nouveau type_ptr>(expression_ptr)`
Force la conversion de type dans le cas de pointeurs
void * p=NULL;
int * r = reinterpret_cast<int *>(p);

Avantages :

- Les conversions sont plus **visibles**.
- **Deux opérateurs** différents pour deux opérations différentes (il y en a deux autres)

Compilation

- Comme en C, le code est séparé en un fichier d'entêtes (.h) et un fichier d'implantation (.cpp).
- L'exécution commence par la fonction
`int main(int argc, char * argv[])` ou
`int main()`
- Compilation avec GNU C++
 - `g++ -Wall fichier.cpp`
compilation et édition de liens, fichier.cpp doit contenir un `main`
 - `g++ -Wall -c fichier.cpp`
compilation seule, crée fichier.o
 - `g++ -Wall fichier1.o fichier2.o -o fichierexe`
édition de liens