

Programmation C++ Première Partie

Michael Mrissa



Département Informatique
UFR Sciences et Techniques
Université de Pau et des Pays de l'Adour

Note

- Ce support est une évolution de celui de Nicolas Belloir
- Il est inspiré des supports de
 - Jean-Michel Bruel (IUT Blagnac)
 - Marc Daniel (Ecole Supérieure d'Ingénieurs de Luminy)

Introduction

Introduction

- Seulement une introduction au langage C++
- Bases de C nécessaires

Historique du langage C++

- développé dans les laboratoires Bell d'ATT par Bjarne Stroustrup en 1980
- basé sur la syntaxe du langage C
- enrichi de mots clés et de mécanismes pour supporter les concepts orientés objet

Plan

- 1 Introduction
- 2 De C à C++
 - Différences entre C et C++
 - Syntaxe
 - Gestion de la mémoire
- 3 Les Fonctions
 - Déclarations
 - Passages de paramètres
- 4 Les Classes et les Objets
 - Définitions
 - Utilisation d'une classe
 - Constructeur et destructeurs
 - L'affectation
 - Un exemple complet : la pile d'entiers

De C à C++

De C à C++

- C++ une amélioration “compatible” de C
 - Permet
 - L'abstraction de données, la programmation objet
 - La généricité
 - ...
 - Beaucoup plus rigoureux que C
 - Fortement typé
 - Langage de référence normalisé
 - Peut être aussi rapide que C (pas de mécanisme sous-jacent complexe)
- Points faibles de C++
 - Compatibilité avec C
 - Langage pouvant être totalement ésotérique
 - Présence de deux syntaxes
 - Utilisation de fonctions C

Un premier programme C++

Exemple

```
#include <iostream>
void main (void) {
    cout << "Hello word" << endl ;
}
```



Les commentaires

Exemple

```
#include <iostream> //commentaire jusqu'en fin de ligne  
// autre commentaire  
/* marche aussi */
```

- Une seule syntaxe pour les commentaires !
- De l'importance
 - Des commentaires
 - Explicites, complets, nombreux
 - Des entêtes
 - Systématiques, homogènes, précis



Les entrées/sorties

- E/S en C : `scanf` et `printf`
- E/S par flot (`iostream`) :
 - `cout` : sortie standard
 - `cin` : entrée standard
 - `cerr` : sortie standard d'erreur non tamponée
 - `clog` : sortie standard d'erreur tamponée
- L'opérateur `<<` permet d'envoyer des valeurs dans un flot de sortie, tandis que `>>` permet d'extraire des valeurs d'un flot d'entrée.



Les entrées/sorties

Exemple

```
#include <iostream.h>

int main() {
    int i=123;
    char ch[80]="Bonjour\n", rep;
    cout << "i=" << i << "   ch=" << ch;
    cout << "i = ? ";
    cin >> i;           // lecture d'un entier
    cout << "rep = ? ";
    cin >> rep;         // lecture d'un caractère
    cout << "ch = ? ";
    cin >> ch;          // premier mot d'une chaîne
    cout << "ch= " << ch; // vérification
    return(0);
}
```

```
/*—résultat de l'exécution —————
i=123 ch=Bonjour
i= ? 12
rep = ? y
ch = ? c++ is easy
ch= c++
*/
```



Les entrées/sorties

- Les principaux intérêts dans l'utilisation des flots :
 - vitesse d'exécution plus rapide
 - vérification de type : pas d'affichage erroné.
 - On peut utiliser les flux avec les types utilisateurs



Définition de variables

- En C++ on peut déclarer les variables ou fonctions n'importe où dans le code. La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.
- Exemple :

Exemple

```
#include <iostream.h>
int main() {
    int j = 0;
    for(int w=0; w<10; w++, j++)
    {
        int i = w;
        cout << i << ' ';
    }
    int k = j; //permet une initialisation "dynamique"
    return(0);
}
```



Visibilité des variables

- L'opérateur de résolution de portée `::` permet d'accéder aux variables globales plutôt qu'aux variables locales.
- Exemple :

Exemple

```
#include <iostream>

int i = 11;

int main() {
    int i = 34;
    {
        int i = 23;
        ::i = ::i + 1; // 11 + 1
        std::cout << ::i << " " << i << std::endl; // affiche 12 23
    }
    std::cout << ::i << " " << i << std::endl; // affiche 12 34
    return 0;
}
```



Les constantes

- Il est possible en C++ de définir une constante en utilisant le mot réservé `const`.
- L'objet ainsi spécifié ne pourra pas être modifié durant toute la durée de sa vie.
- Il est indispensable d'initialiser la constante au moment de sa définition.
- Exemple :

Exemple

```
const int N = 10;    // entier constant  
int tab[2 * N];     // #define aussi
```



Différence entre const et #define

- Le #define est une directive pour le préprocesseur
- Ce dernier remplace par ce qui est défini AVANT le compilateur compile
- const est lié à une variable, on pourra
 - la caster
 - récupérer son adresse
 - pointer dessus
 - la convertir, etc.
- et on bénéficie des avantages d'une variable (type checking, debugging, portée...)



Les types composés

- **typedef** n'est plus obligatoire pour renommer un type.
- Exemple :

Exemple

```
struct FICHE {  
    char *nom, *prenom;  
    int age;  
};  
FICHE adherent, *liste;
```



Les variables références

- Le C++ offre les variables références
- Référence = variable “synonyme” d’une autre
- Toute modification de l’une affectera le contenu de l’autre
- Une variable référence doit obligatoirement être initialisée et le type de l’objet initial doit être le même que l’objet référence



Les variables références

Exemple

```
int i;  
int &ir = i; // ir référence à i  
int *ptr;  
  
i=1;  
cout << "i= " << i << " ir= " << ir << endl;  
// affichage de : i= 1 ir= 1  
ir=2;  
cout << "i= " << i << " ir= " << ir << endl;  
// affichage de : i= 2 ir= 2  
ptr = &ir;  
*ptr = 3;  
cout << "i= " << i << " ir= " << ir << endl;  
// affichage de : i= 3 ir= 3
```



Les variables références

- Trois différences avec les pointeurs :
 - Initialisation obligatoire
 - Toute opération sur la référence agit sur l'objet référencé (et non pas sur l'adresse)
 - la valeur de référence ne peut pas être modifiée



L'allocation mémoire

- Deux opérateurs `new` et `delete` pour remplacer respectivement les fonctions `malloc` et `free`.
- L'opérateur `new` réserve l'espace mémoire qu'on lui demande et l'initialise
- L'opérateur `delete` libère l'espace mémoire alloué par `new` à un seul objet, tandis que l'opérateur `delete[]` libère l'espace mémoire alloué à un tableau d'objets.



Exemple

```
int *ptr1, *ptr2, *ptr3;  
// allocation dynamique d'un entier :  
ptr1 = new int;  
// allocation d'un tableau de 10 entiers :  
ptr2 = new int [10];  
// allocation d'un entier avec initialisation  
ptr3 = new int(10);  
  
struct date {int jour, mois, an; };  
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};  
// allocation dynamique d'une structure :  
ptr4 = new date;  
// allocation dynamique d'un tableau de structure  
ptr5 = new date[10];  
// allocation dynamique d'une structure avec init.  
ptr6 = new date(d);
```



Les Fonctions

Déclarations

- En C++, les déclarations de fonctions sont identiques aux prototypes de fonctions de la norme **C-ANSI**. Le programmeur doit déclarer le nombre et le type des arguments de la fonction.
- Par contre, cette déclaration est obligatoire avant utilisation.



Passage par valeur

- Une **copie** de l'argument est passé à la fonction
- Ne modifie pas l'original

Exemple

```
#include <iostream>
void echange(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
int main() {
    int i=2, j=3;
    echange(i, j);
    cout << "i=" << i << " j=" << j << endl;
    // affichage de : i= 2 j= 3
    return (0);
}
```



Passage par pointeur (ou par adresse)

- L'**adresse** de l'argument est passé à la fonction
- La fonction manipule les adresses

Exemple

```
#include <iostream>
void echange(int * n1, int * n2) {
    int temp = * n1;
    *n1 = *n2;
    *n2 = temp;
}
int main() {
    int i=2, j=3;
    echange(&i, &j);
    cout << "i=" << i << " j=" << j << endl;
    // affichage de : i= 3 j= 2
    return (0);
}
```



Passage par référence

- C++ définit le passage par référence.
- Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un "synonyme" du paramètre réel.
- Toute modification du paramètre référence est répercutée sur le paramètre réel.

Exemple

```
#include <iostream>
void echange(int &n1, int &n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
int main() {
    int i=2, j=3;
    echange(i, j);
    cout << "i=" << i << " j=" << j << endl;
    // affichage de : i= 3 j= 2
    return (0);
}
```



Bilan

- **Favoriser le passage par référence**
 - Plus lisible
- Ajout du mécanisme `const` à un paramètre
 - Signale au compilateur qu'il ne doit pas modifier des valeurs recopiées ou signalées comme non modifiables
 - Sécurité des programmes, aide à la correction
 - **Excellente habitude**
- Ne pas confondre
 - `const type * ptr` : ptr est un pointeur sur un objet constant
 - `type * const ptr` : ptr est un pointeur constant sur un objet type!!!



Valeur par défaut des paramètres

- Certains arguments d'une fonction peuvent prendre souvent la même valeur. Pour ne pas avoir à spécifier ces valeurs à chaque appel de la fonction, le C++ permet de déclarer des valeurs par défaut dans le prototype de la fonction.
- Remarque : les paramètres par défaut sont obligatoirement les derniers de la liste. Ils ne sont déclarés que dans le prototype de la fonction et pas dans sa définition.

Exemple

```
#include <iostream>
void bidon(int, float=10.0, int=10) { //ou void bidon(int, float f=10.0, int j=10)
    cout << "je porte bien mon nom" << m << " " << f << " " << j;
}
int main() {
    bidon (1, 1.0, 2);
    bidon (1, 1.0);
    bidon (1);
    return (0);
}
```

Fonctions *en ligne*

- Une fonction peut être définie **inline**.
- Lorsqu'une fonction est spécifiée **inline**, le compilateur essaye de générer un code expansé en ligne pour l'appel de la fonction (et par exemple remplacer dans le code **carre(2)** par **4**).
- Une fonction en ligne doit être définie dans le fichier source où elle est utilisée.
- Exemple :

Exemple

```
inline carre(int i) { return i*i; }
```



Les Classes et les Objets



Définition

- Une classe en C++ est une structure qui contient :
 - des fonctions membres
 - des données membres
- Les mots réservés **public**, **private** et **protected** délimitent les sections visibles par l'application.



Exemple

```
#include <iostream>
class Avion {
public : // _____
    void init(char [], char *, float);
    void affiche();
private : // _____
    char immatriculation[6], *type; /* données membres privées */
    float poids;
    void erreur(char *message); /* fonction membre privée */
}; // attention à ne pas oublier ce ;
```



Définition

- Une classe est une unité de compilation
 - Un fichier entête (Classe.h) qui définit la classe (appelé aussi interface)
 - Un fichier source (Classe.cxx) qui contient l'implémentation des fonctions membres
 - L'implémentation des fonctions peut se faire dans l'entête. Les fonctions sont alors inline par défaut.
 - Se justifie pour des fonctions très courtes
 - Une fonction implémentée dans (Classe.cxx)
 - Peut être inline
 - Doit être qualifiée par le nom de la classe par l'opérateur de résolution de portée ::
 - `type Classe::fonction_membre(signature){...};`



Définition

- Un objet est une instance d'une classe
- Exemple :
 - `Avion airbus, boeing; // 2 instances`



Droits d'accès

- L'**encapsulation** consiste à masquer l'accès à certains attributs et méthodes d'une classe. Elle est réalisée à l'aide des 3 mots clés :
 - **private** : les membres privés ne sont accessibles que par les fonctions membres de la classe.
 - **protected** : les membres protégés sont comme les membres privés mais ils sont aussi accessibles par les fonctions membres des classes dérivées
 - **public** : les membres publics sont accessibles par tous (par défaut).



Droits d'accès et types de classes

- Il existe différents types de classes :
 - `struct Classe1 /* ... */;` : accès aux membres **public** par défaut, pas d'accesseurs ni de mutateurs, toute la structure est en mémoire
 - `union Classe2 /* ... */;` : accès aux membres **public** par défaut, pas d'accesseurs ni de mutateurs, mémoire réservée pour le plus grand membre de l'union, un seul membre de l'union est censé être valide à un instant t
 - `class Classe3 /* ... */;` : accès aux membres **private** par défaut, données membres en mémoire, accesseurs et mutateurs publics
- C'est cette dernière forme qui est utilisée en C++ pour définir des classes.



Définition des fonctions membres

- En général, la déclaration d'une classe contient simplement les prototypes des fonctions membres de la classe

Exemple

```
class Avion {  
    public :  
        void init(char [], char *, float);  
        void affiche();  
    private :  
        char _immatriculation[6], *_type;  
        float _poids;  
        void _erreur(char *message);  
};
```

Fonctions membres

- Les fonctions membres sont définies dans un module séparé ou plus loin dans le code source

Exemple

```
void Avion::init(char m[], char *t, float p) {  
    strcpy(_immatriculation, m);  
    _type = new char [strlen(t)+1];  
    strcpy(_type, t);  
    _poids = p;  
}
```



Instantiation d'une classe

- Le nom d'une classe représente un nouveau type de donnée.

Exemple

```
Avion av1;      // une instance simple  
Avion compagnie[10]; // un tableau  
Avion *av2; // un pointeur non initialisé  
av2 = new Avion; // création dynamique
```



Instantiation d'une classe

- Deux opérateurs

Exemple

```
Avion av1;           // une instance simple
Avion *av2; // un pointeur non initialisé
av2 = new Avion; // création dynamique
Avion *av3 = new Avion[3]; //tableau
delete av2; // libération mémoire
delete [] av3;
```



Utilisation des objets

- Après avoir créé une instance on peut accéder aux attributs et méthodes de la classe.
- Utiliser l'opérateur `.` (membre d'instance) ou `->` (membre de pointeur).

Exemple

```
av1.init("FGBCD", "TB20", 1.47);  
av2->init("FGDEF", "ATR 42", 80.0);  
compagnie[0].init("FEFGH", "A320", 150.0);  
av1.affiche();
```



Le pointeur `this`

- Lorsque l'on appelle une méthode d'une classe, celle-ci reçoit en plus de ses paramètres, un paramètre caché : le pointeur `this`. Ce pointeur (constant) permet à la méthode d'accéder à l'objet qui l'a appelée

Exemple

```
class X {  
    public:  
        int f() { return this->i; }  
}
```



Notions de constructeur

- Les données membres d'une classe doivent être initialisées par une méthode d'initialisation
- Le **constructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à l'instanciation de l'objet.

Exemple

```
class Cercle {  
    int x, y; int rayon;  
    public :  
        Cercle(int , int=0, int=0); // constructeur  
};  
Cercle::Cercle(int r, int cx, int cy) {  
    rayon = r; x = cx; y = cy; }  
  
Cercle ballon(20,10,10);
```



Notions de destructeur

- De même, après avoir fini d'utiliser un objet, il est bon de prévoir une méthode permettant de détruire l'objet (en libérant par exemple la mémoire)
- le **destructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet. Ce destructeur est une fonction qui porte comme nom, le nom de la classe précédé du caractère `~` (tilda), qui ne retourne pas de valeur (pas même un **void**) et qui n'accepte aucun paramètre (le destructeur ne peut donc pas être surchargé)

L'affectation

- L'affectation peut intervenir dans plusieurs cas :

Exemple

```
Elements p = Elements(10,20);  
Elements p0(10,20);  
Elements p1,p2,*p3;  
...  
p1=p2;  
f(p1);  
p3 = new Elements(20,30);  
delete p3;  
p3 = &p1;
```

L'affectation

- On peut contrôler ce qui est fait :

Exemple

```
void operator=(const Element&); // affectation  
Element(const Element&); // initialisation
```



Le fichier IntStack.h

Exemple

```
// IntStack.h
#ifndef IntStack_h
#define IntStack_h

class IntStack {
public:
    void init(int taille = 10); // création d'une pile
    void push(int n); // empile un entier au sommet de pile
    int pop(); // retourne l'entier au sommet de pile
    int vide() const; // vrai, si la pile est vide
    int pleine() const; // vrai, si la pile est pleine
    int getsize() const { return _taille; }
private:
    int _taille; // taille max de la pile
    int _sommet; // position de l'entier à empiler
    int *_addr; // adresse de la pile
};
#endif
```



Le fichier IntStack.cpp

Exemple

```
//IntStack.cc
#include "IntStack.h"
#include <cassert>           // pour debugger

void IntStack::init(int taille) {
    _addr = new int [ _taille = taille ];
    assert( _addr != 0 ); // au cas où new plante
    _somet = 0;
}

void IntStack::push(int n) {
    if ( ! pleine() )
        _addr[ _somet++ ] = n;
}

int IntStack::pop() {
    return ( ! vide() ) ? _addr[ --_somet ] : 0;
}

int IntStack::vide() const {
    return ( _somet == 0 );
}

int IntStack::pleine() const {
    return ( _somet == _taille );
}
```

Le fichier MainIntStack.cpp

Exemple

```
//main.cc
#include "IntStack.h"
#include <iostream>
#include <cstdlib>      // pour utiliser rand()

int main() {
    IntStack pile;
    pile.init(15);      // pile de 15 entiers
    while ( ! pile.pleine() ) // remplissage de la pile
        pile.push( rand() % 100 );
    while ( ! pile.vide() ) // Affichage de la pile
        std::cout << pile.pop() << " ";
    std::cout << std::endl;
    return 0;
}
```

