Programmation Orientée Objet

Théorie C++

Jean-Marc Wagner
(Notes de cours : C. Vilvens)

Table des matières

Table des matières

Les commentaires

- En C : /* ... */
- C++ utilise également le symbole // : signifie début du commentaire jusqu'au retour de chariot

Exemple :

```
// Ceci est un commentaire
```

qui peut aussi s'écrire :

```
/* Ceci est un commentaire */
```

Table des matières

Les flux

L'approche des E/S du C++ vise deux objectifs :

- La *souplesse* : une donnée d'un type donné doit être traité avec son type → les conversions nécessaires sont automatiques
- L'*extensibilité* : le système d'E/S doit pouvoir s'étendre
  - à de nouveaux types de données définis par le programmeur
  - à de nouveaux paradigmes d'E/S. Exemple : un flux réseau

Tous les éléments d'E/S sont des *flux* (ou *streams*) sur lesquels on *insère* ou/et on *extrait* des données.

Les flux

Trois flux prédéfinis dans la librairie standard du C++ :

- cout : flux de sortie associé à l'écran (le stdout du C)
- cin : flux d'entrée associé au clavier (le stdin du C)
- cerr : flux d'erreur associé à l'écran (le stderr du C)

qui sont déclarés dans <stream.h>, <iostream.h> ou <iostream> selon le système.

Le flux de sortie cout

Afficher une donnée revient donc à l'insérer sur le flux de sortie cout → utilisation de l'opérateur d'insertion << :

```
cout << {variable|constante|expr.} [<< {variable|constante|expr.}] [...]
```

En pratique :

on écrit en C	on écrit en C++
printf("%d",n);	cout << n;
printf("%f",fNb);	cout << fNb;
printf("%s",msg);	cout << msg;
printf("Resultat = %d",res);	cout << "Resultat = " << res;

HELLO01.cxx

Notre premier programme C++ :

```
// Hello01.cxx
#include <stream.h>

int main()
{
   cout << "Hello, World !";
}
```

Pour compiler sur Sunray :

g++ Hello01.cxx

ou encore :

g++ Hello01.cxx -o Hello01

Le flux d'entrée cin

Lire une donnée au clavier revient à l'extraire du flux d'entrée cin → utilisation de l'opérateur d'extraction >> :

```
cin >> variable [ >> variable] [...]
```

En pratique :

on écrit en C	on écrit en C++
scanf("%d",&n);	cin >> n;
scanf("%f",&fNb);	cin >> fNb;
char msg[20]; scanf("%s",msg);	cin >> msg;

Le flux d'entrée cin (HELLO03.cxx)

```
// HELLO03.CXX (Sunray)
#include <iostream>
using namespace std;

int main()
{
   char name[20];
   cout << "Hello, World !" "\nTell me what's your name ? ";
   cin >> name;
   cout << "\nHello " << name << "!\n";
}
```

Un manipulateur (HELLO04.cxx)

Plutôt que d'utiliser explicitement le caractère "\n", on insère le *manipulateur* endl sur le flux de sortie :

```
// HELLO04.CXX (Sunray)
#include <iostream>
using namespace std;

int main()
{
   char name[20];
   cout << "Hello, World !" << endl << "Tell me what's your name ? ";
   cin >> name;
   cout << endl << "Hello " << name << "!" << endl;
}
```

Table des matières

La déclaration des variables

- En C, les variables doivent être déclarées en début de bloc
- En C++, on peut déclarer une variable au moment précis où l'on va s'en servir pour la première fois

```
int main() // DECLAVA3.CXX
{
   int nE;
   cout << "Nombre d'employes : "; cin >> nE;
   for (int i=0;i<nE;i++)
   {
      char nom[20];
      cout << "Nom : "; cin >> nom;
      float sal;
      cout << "Salaire : "; cin >> sal;
      ...
   }
   cout << "valeur de i finale : " << i << endl; // ATTENTION !!
}
```

La déclaration des variables

- En C++, le mot réservé struct ne doit plus être répété lorsque l'on déclare une variable du type de cette structure

```
// DECLAVA4.CXX
struct agent
{   char nom[20];
    float sal;
    int anc; };

int main()
{
   ...
   agent a; // pas de mot struct
   cout << "Nom : "; cin >> a.nom;
   cout << "Salaire : "; cin >> a.sal;
   cout << "Anciennete : "; cin >> a.anc;
   ...
}
```

Table des matières

L'allocation dynamique

- En C, les allocations et libérations de mémoire se font au moyen des fonctions : malloc() et free()
- En C++, on utilise plutôt des *opérateurs de création* et *de destruction*

Syntaxe :

```
new type [ [taille memoire] ]
```

qui renvoie un pointeur sur une mémoire du type désiré

```
delete [ [] ] pointeur
```

qui libère l'espace mémoire indiqué par le pointeur

En pratique :

on écrit en C	on écrit en C++

int *pE = (int*) malloc(sizeof(int));
*pE = 5;
free(pe);
	int *pE = new int;
	*pE = 5;
	delete pE;

char *msg = (char*) malloc(20);	char *msg = new char[20];
float *aReal = (float*) calloc(5,sizeof(float)); free (aReal);	float *aReal = new float[5]; delete [ ] aReal;

struct agent {
char nom[20];
int persCh;
float sal; }
struct agent *pAgent;
pAgent = (struct agent *)
malloc(sizeog(struct agent));
	struct agent {
	char nom[20];
	int persCh;
	float sal; }
	agent *pAgent;
	pAgent = new agent;

Table des matières

Les fonctions sans argument

En C,

```
int fBof();
```

signifie que la fonction admet un nombre quelconque de paramètres de type quelconque.

```
int fRien(void);
```

signifie que la fonction n'admet aucun paramètre.
En C++,

```
int fBof();
```

signifie que *la fonction n'admet aucun paramètre* !

Accéder à une variable globale

- utilisation de l'*opérateur de résolution de portée* : "::"

```
// LOCGLOB1.CXX (Sunray)
#include <iostream>
using namespace std;

int i =120;

int main ()
{
   int i = -9;
   cout << "i = " << i << endl;    // var. locale
   cout << "i = " << ::i << endl;  // var. globale
}
```

Les fonctions inline

- Ce sont des fonctions
  - qui allient l'avantage des macros (pas de véritable appel généré)
  - qui ont l'attrait de fonctions classiques (analyse par le compilateur, paramètres typés, ...)
- Pour cela, on utilise le *mot réservé* "inline"
- il ne s'agit donc pas d'un appel de fonction au sens machine du terme
- N'a d'intérêt que si son code est de petite taille

D'une version macro (INLINE0.cxx)...

```
#define dis0(x,y) sqrt ( (x)*(x) + (y)*(y) )
double Distance (double , double , double , double );

int main()
{
   float xa, ya;
   ...
   cout << "distance a l'origine = " << dis0(xa,ya) << endl;

   float xb, yb;
   ...
   cout << "distance des deux points = " << distance(xa,ya,xb,yb) << endl;
}

double Distance (double x1, double y1, double x2, double y2) {
   double dx = x2 - x1;
   double dy = y2 - y1;
   return sqrt( dx*dx + dy*dy ); }
```

...à une version inline (INLINE1.cxx)

```
inline double dis0 (double x, double y) {
   return sqrt( x*x + y*y ); }

inline double Distance (double x1, double y1, double x2, double y2) {
   double dx = x2 - x1;
   double dy = y2 - y1;
   return sqrt( dx*dx + dy*dy ); }

int main()
{
   float xa, ya;
   ...
   cout << "distance a l'origine = " << dis0(xa,ya) << endl;

   float xb, yb;
   ...
   cout << "distance des deux points = " << Distance(xa,ya,xb,yb) << endl;
}
```

Les arguments par défaut

- En C++, il est possible de fournir des valeurs par défaut aux derniers paramètres d'une fonction
- Pour cela, il suffit de préciser ces valeurs *dans la déclaration* de la fonction
- Restriction : les arguments à valeur par défaut doivent être *cités impérativement les derniers*

Exemple :

```
double Distance(double x1,double y1,double x2=0,double y2=0);
```

Les arguments par défaut (DEFAUT1.cxx)

```
double Distance(double x1, double y1, double x2=0, double y2=0);

int main()
{
   float xa, ya;
   ...
   cout << "distance a l'origine = " << Distance(xa,ya) << endl;

   float xb, yb;
   ...
   cout << "distance des deux points = " << Distance(xa,ya,xb,yb) << endl;
}

double Distance(double x1, double y1, double x2, double y2) {
   double dx = x2 - x1;
   double dy = y2 - y1;
   return sqrt( dx*dx + dy*dy );
}
```

Table des matières

Paramètres passés par adresses (rappel)

- Pour qu'une fonction puisse modifier un de ses paramètres, il faut que ce paramètre soit passé par adresse → on utilise un pointeur

Exemple :

```
void permute(float *a,float *b)
{
   float tr = *a;
   *a = *b;
   *b = tr;
}
```

dont l'appel pour deux réels x et y serait :

```
permute(&x,&y);
```

→ le programmateur qui utilisera cette fonction doit savoir que les paramètres sont passés par adresse...

Paramètres passés par références

C++ propose un autre mode de passage de paramètres, appelé "*passage par référence*".

Une *référence*, tout comme un pointeur, désigne un *emplacement mémoire*. Mais, au contraire d'un pointeur, cet emplacement se désigne seulement par son nom, sans devoir utiliser l'opérateur *

Par conséquent, un paramètre passé par référence

- s'utilise seulement par son nom
- désigne l'emplacement du paramètre effectif qui peut donc être modifié

Syntaxe :

```
type & variable_reference
```

Paramètres passés par références

La fonction permute() peut à présent s'écrire :

```
void permute(float& a,float& b)
{
   float tr = a;
   a = b;
   b = tr;
}
```

L'appel sera simplement :

```
permute(x,y);
```

Lors du passage de paramètres, a référence x et b référence y : ils désignent donc le même emplacement mémoire.

Paramètres passés par référence (REFEREN1.cxx)

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

void permute (float& a,float& b);

int main()
{
   float x=2.23,y(5.25);

   cout << "x et y = " << x << " et " << y << endl;
   permute(x,y);
   cout << "x et y = " << x << " et " << y << endl;

   permute(12.3,x);  // gloups !
   cout << "x = " << x << endl;
}

void permute (float& a,float& b) {
   float tr = a;
   a = b;
   b = tr; }
```

Fonction renvoyant une référence (REFEREN2.cxx)

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

float& permute (float& a, float & b);

int main()
{
   float x=2.23, y(5.25);
   cout << "x et y = " << x << " et " << y << endl;
   cout << "Moyenne = " << permute(x,y) << endl;    // rvalue
   cout << "x et y = " << x << " et " << y << endl;
   cout << (permute(x,y) = 4.0) << endl;            // lvalue
   cout << "x et y = " << x << " et " << y << endl;
}

float & permute (float& a, float & b) {
   float tr = a;
   a = b; b = tr;
   float * m = new float;  // liberation memoire ???
   *m = (a + b) / 2;
   return *m; }
```

Des références indépendantes (REFEREN3.cxx)

La déclaration de références peut également se faire en dehors de la liste d'arguments d'une fonction. On peut donc imaginer ceci (mais cela se fait rarement) :

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

int main()
{
   int x =10, y = 50;

   int & rI = x;
   cout << "x, y et rI = " << x << "," << y << " et " << rI << endl;

   rI = y; y = 160;
   cout << "x, y et rI = " << x << "," << y << " et " << rI << endl;
}
```

Des références indépendantes

Dès lors,

- Une référence doit être *initialisée à sa déclaration*
- Une référence ne peut jamais désigner une autre variable que celle de sa définition; autrement dit, *elle ne peut être modifiée*

Table des matières

Les passages de paramètres

Quel mode de passage de paramètres choisir ?

- Si une fonction ne modifie pas un argument de type prédéfini : le passer par valeur
- Si une fonction modifie un argument de type prédéfini : le passer par adresse
- Si une fonction modifie un argument d'un type défini par l'utilisateur : le passer par référence
- Si une fonction ne modifie pas un argument d'un type défini par l'utilisateur et de grande taille : le passer par référence

Table des matières

Le polymorphisme des fonctions

C++ permet à des fonctions différentes de porter le même nom. Plus exactement,

- C++ permet à une fonction d'avoir *plusieurs versions différentes* : on parle encore du *polymorphisme* des fonctions
- Une version d'une fonction diffère d'une autre version *par le type d'au moins un paramètre*. On parle encore de "*signature*" des fonctions
- *Le type de la valeur renvoyée n'intervient pas du tout* dans la distinction entre deux fonctions de même nom

Exemple :

```
double moyenne(double x);
long moyenne(long x);
```

L'appel

```
moyenne(n);
```

dépend du type de n.

Table des matières

La déclaration des constantes

En C, on utilise habituellement des macros :

```
#define NE 20
#define genie "Vilvens"
```

Ces constantes

- n'ont pas d'adresse
- ne peuvent être que d'un type simple prédéfini
- ont une portée allant de leur déclaration à la fin du fichier source

La déclaration des constantes

En C++, on déclarera une constante selon la syntaxe :

```
const type identificateur = valeur
```

On écrira donc :

```
const int NE = 20;
const char* genie = "Vilvens"
const float vecF[2] = { 12.36 , 15.965}
```

Ces fois, les constantes

- ont une adresse (un pointeur peut les désigner)
- peuvent être d'un type quelconque
- ont une portée analogue à celle des variables

Caractéristiques des constantes C++

- Les constantes du C++ peuvent être de “vraies constantes”

```
const int NE = 20;
float vec[NE];
```

- Une constante du C++ doit toujours être initialisée
- La portée d'une constante définie en dehors de toute fonction est limitée au module qui la contient

Les pointeurs de constantes et les pointeurs constants

- Dans la déclaration suivante :

```
const int *pE;
```

pE pointe un entier constant : la variable *pE ne peut varier. Mais le pointeur pE peut varier.

- Par contre :

```
int * const pE;
```

le pointeur pE est constant. La valeur pointée *pE peut varier.

Des paramètres constants (CONST1.cxx)

On peut passer une donnée pointée à une fonction en assurant qu'elle ne peut modifer cette donnée :

```
void AfficheNom(const char * n);
```

Le compilateur détectera une erreur dans le programme suivant :

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

void afficheNom (const char *n);

int main()
{
   char * myName = "Vilvens";
   afficheNom(myName);
}

void afficheNom (const char *n) {
   cout << n << endl;
   strcpy(n,"Claude"); // pose probleme...
}
```

Des paramètres constants (CONST2.cxx)

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

void machin (int & n);
void machin (const int & n);

void machin (int & n) {
   cout << ++n << endl; }

void machin (const int & n) {
   cout << n << endl; }
...

int main()
{
   int x(23), y;
   machin(x);
   cout << "Un nombre : " << endl;
   cin >> y;
   machin(y);
   ...
}
```