Cours Langage C++

Thierry Vaira

IUT Arles

tvaira@free.fr

Historique (1/3)

- En 1970, **Ken Thompson**, créa un nouveau langage : Le B, descendant du BCPL (*Basic Combined Programming Language*, créé en 1967 par Martin Richards). Son but était de créer un langage simple, malheureusement, son langage fût trop simple et trop dépendant de l'architecture utilisée.
- En 1971, **Dennis Ritchie** commence à mettre au point le successeur du B, le **C**. Le résultat est convaincant : Le C est **totalement portable** (il peut fonctionner sur tous les types de machines et de systèmes), il est de **bas niveau** (il peut créer du code aussi rapide que de l'assembleur) et il permet de traiter des **problèmes de haut niveau**. Le C permet de quasiment tout faire, du driver au jeu. Le C devient très vite populaire.

Historique (2/3)

- En 1989, l'**ANSI** (*American National Standards Institute*) normalisa le C sous les dénominations **ANSI C** ou **C89**. Un programme écrit dans ce standard est compatible avec tous les compilateurs.
- En 1983, **Bjarne Stroustrup** des laboratoires Bell crée le **C++**. Il construit donc le C++ sur la base du C. Il garde une forte compatibilité avec le C.
- En 1999, l'**ISO** (*International Organization for Standardization*) proposa une nouvelle version de la norme, qui reprenait quelques bonnes idées du langage C++. Il ajouta aussi le type `long long` d'une taille minimale de 64 bits, les types complexes, l'initialisation des structures avec des champs nommés, parmi les modifications les plus visibles. Le nouveau document est celui ayant autorité aujourd'hui, est connu sous le sigle **C99**.

Historique (3/3)

- Certaines autres extensions du C ont elles aussi été standardisées, voire normalisées. Ainsi, par exemple, des fonctions spécifiques aux systèmes **UNIX**, sur lesquels ce langage est toujours très populaire, et qui n'ont pas été intégrées dans la norme du langage C, ont servi à définir une partie de la norme **POSIX**.
- Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (**C++98**), puis une seconde en 2003. Le standard actuel a été ratifié et publié par ISO en septembre 2011 (**C++11**). Mais certains compilateurs ne la supportent pas encore complétement.
- Les **langages C et C++ sont les langages les plus utilisés dans le monde de la programmation**.

Apports du C++ par rapport au C

Le C++ a apporté par rapport au langage C les notions suivantes :

- les concepts orientés objet (encapsulation, héritage),
- les références, la vérification stricte des types,
- les valeurs par défaut des paramètres de fonctions,
- la surcharge de fonctions (plusieurs fonctions portant le même nom se distinguent par le nombre et/ou le type de leurs paramètres),
- la surcharge des opérateurs (pour utiliser les opérateurs avec les objets), les constantes typées,
- la possiblité de déclaration de variables entre deux instructions d'un même bloc

C vs C++

- On peut passer progressivement de C à C++.
- Il suffit en premier lieu de **changer de compilateur (par exemple `gcc` → `g++`)**, sans nécessairement utiliser les nombreuses possibilités supplémentaires qu'offre C++.
- Le principal intérêt du passage de C à C++ est de profiter pleinement de la puissance de la **programmation orientée objets (POO)**.

L'option `-std` des compilateurs `gcc`/`g++` permet de choisir la norme à appliquer au moment de la compilation. Par exemple : `-std=c89` ou `c99` pour le C et `-std=c++98` ou `c++0x` pour le C++ (compilateur version 4.4.3)

Exemple : premier programme en C++

```
#include <iostream> // pour cout, cin

int main (int argc, char **argv)
{
   int n;

   std::cout << "Donnez un entier : " << std::endl;
   std::cin >> n;

   for(int i = 0; i < n; i++)
      std::cout << "Hello world !" << std::endl;

   return 0;
}
```

L'extension par défaut des fichiers écrits en langage C++ est .cpp ou .cc

- Par défaut, gcc et g++ fabriquent un exécutable de nom a.out. Sinon, on peut lui indiquer le nom du fichier exécutable en utilisant l'option -o.

Fabrication de l'exécutable (sous Linux)

```
$ g++ <fichier.cpp> -o <executable>
```

L'exécution du programme donne le résultat suivant :

```
$ ./a.out
Donnez un entier : 2
Hello world !
Hello world !
```

Étapes de fabrication

1. Le préprocesseur (pré-compilation)
   - Traitement de toutes les directives `#xxxxx`
   - Inclusion de fichiers (`.h`)
   - Substitutions lexicales : les "macros"
2. La compilation
   - Vérification de la syntaxe
   - Traduction dans le langage d'assemblage de la machine cible
3. L'assemblage
   - Traduction finale en code machine
   - Production d'un fichier objet (`.o` ou `.obj`)
4. L'édition de liens
   - Unification des symboles internes
   - Étude et vérification des symboles externes (bibliothèques `.so` ou `.DLL`)
   - Production du programme exécutable

Étapes de fabrication avec g++ (1/2)

Pré-compilation

```
$ g++ -E <fichier.cpp> -o <fichier_precompile.cpp>
```

Compilation

```
$ g++ -S <fichier_precompile.cpp> -o <fichier.s>
// ou
$ g++ -c <fichier.cpp> -o <fichier.o>
```

Assemblage

```
$ as <fichier.s> -o <fichier.o>
```

Étapes de fabrication avec g++ (1/2)

Édition des liens

```
$ g++ <fichier.o> -o <executable>
// ou
$ g++ <fichier1.o> <fichier2.o> <fichiern.o> -o <executable>
// voir aussi : $ ld -dynamic-linker <fichier.o> -o <executable>
```

Fabrication

```
// Combiner en une seule ligne de commande toutes les étapes (préprocesseur,
    compilation, assemblage et édition des liens)
$ g++ <fichier.cpp> -o <executable>
```

Décomposition des étapes

```
$ g++ -v -save-temps <fichier.cpp> -o <executable>
```

Définition

- Une variable est un **espace de stockage pour un résultat**.
- Une variable est un **symbole** (habituellement un **nom** qui sert d'identifiant) qui renvoie à une **position de mémoire** (**adresse**) dont le contenu peut prendre successivement différentes valeurs pendant l'exécution d'un programme.

Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes. (Rob Pike)
Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées.» ...

Rappel : La convention de nommage des fichiers pour le C++ est d'utiliser l'extension `.cc` ou `.cpp`.

Exemple : des variables de différents types en C++

```
#include <iostream> /* pour cout */
using namespace std;

int main() /* la fonction principale appelée automatiquement au lancement de l'exécutable */
{
   bool reussie = true; // true est une valeur booléenne
   int nombreDOeufs = 3; // 3 est une valeur entière
   unsigned long int jeSuisUnLong = 12345678UL; // U pour unsigned et L pour long
   float quantiteDeFarine = 350.0; // ".0" rend la valeur réelle
   char unite = 'g'; // ne pas oublier les quotes : ' '

   cout << "La variable nombreDOeufs a pour valeur " << nombreDOeufs << endl;
   cout << "La variable jeSuisUnLong a pour valeur " << jeSuisUnLong << endl;
   cout << "La variable quantiteDeFarine a pour valeur " << quantiteDeFarine << endl;
   cout << "La variable unite a pour valeur " << unite << endl;
   cout << "Recette " << reussie << " : il faut " << nombreDOeufs << " oeufs et " << quantiteDeFarine << "
         " << unite << " de farine" << endl;

   return 0; /* fin normale du programme */
}
```

Les types entiers

- `bool` : false ou true → booléen (seulement en C++)
- `unsigned char` : 0 à 255 ($2^8 - 1$) → entier très court (1 octet ou 8 bits)
- `[signed] char` : -128 (-2^7) à 127 ($2^7 - 1$) → idem mais en entier relatif
- `unsigned short [int]` : 0 à 65535 ($2^{16} - 1$) → entier court (2 octets ou 16 bits)
- `[signed] short [int]` : -32768 (-2^{15}) à +32767 ($2^{15} - 1$) → idem mais en entier relatif
- `unsigned int` : 0 à 4.295e9 ($2^{32} - 1$) → entier sur 4 octets ; **taille "normale" actuelle**
- `[signed] int` : -2.147e9 (-2^{31}) à +2.147e9 ($2^{31} - 1$) → idem mais en entier relatif
- `unsigned long [int]` : 0 à 4.295e9 → entier sur 4 octets ou plus ; sur PC identique à "int" (hélas...)
- `[signed] long [int]` : -2.147e9 à -2.147e9 → idem mais en entier relatif
- `unsigned long long [int]` : 0 à 18.4e18 ($2^{64} - 1$) → entier (très gros !) sur 8 octets sur PC
- `[signed] long long [int]` : -9.2e18 (-2^{63}) à -9.2e18 ($2^{63} - 1$) → idem mais en entier relatif

Les types à virgule flottante

- `float` : Environ 6 chiffres de précision et un exposant qui va jusqu'à $\pm 10^{\pm 38}$ $\rightarrow$ Codage IEEE754 sur 4 octets
- `double` : Environ 10 chiffres de précision et un exposant qui va jusqu'à $\pm 10^{\pm 308}$ $\rightarrow$ Codage IEEE754 sur 8 octets
- `long double` $\rightarrow$ Codé sur 10 octets

Les constantes

- Celles définies **pour le préprocesseur** : c'est simplement une **substitution syntaxique pure** (une sorte de copier/coller). Il n'y a aucun typage de la constante.

```
#define PI 3.1415 /* en C traditionnel */
```

- Celles définies **pour le compilateur** : c'est une **valeur typée**, ce qui permet des contrôles lors de la compilation.

```
const double pi = 3.1415; // en C++ et en C ISO
```

Les pointeurs

- Les pointeurs sont des variables spéciales permettant de **stocker une adresse** (pour la manipuler ensuite).
- L'adresse représente généralement l'**emplacement mémoire d'une variable** (ou d'une autre adresse).
- Comme la variable a un type, le pointeur qui stockera son adresse doit être du même type pour la manipuler convenablement.
- Le type `void*` représentera un **type générique** de pointeur : en fait cela permet d'indiquer sagement que l'on ne sait pas encore sur quel type il pointe.

Utilisation des pointeurs

- On utilise l'étoile (*) pour **déclarer un pointeur**.

  déclaration d'un pointeur (*) sur un entier (int) : `int *ptrInt;`

- On utilise le & devant une variable pour **initialiser ou affecter un pointeur avec une adresse**.

  déclaration d'un entier i qui a pour valeur 2 : `int i = 2;`
  affectation avec l'adresse de la variable i (&i) : `ptrInt = &i;`

- On utilise l'étoile devant le pointeur (*) pour **accéder à l'adresse stockée**.

  indirection ("pointe sur le contenu de i") : `*ptrInt = 3;`

Maintenant la variable i contient 3

Les références en C++

- En C++, il est possible de déclarer une référence `j` sur une variable `i` : cela permet de créer un **nouveau nom `j` qui devient synonyme de `i`** (**un alias**).
- On pourra donc modifier le contenu de la variable en utilisant une référence.
- La déclaration d'une référence se fait en précisant le type de l'objet référencé, puis le symbole &, et le nom de la variable référence qu'on crée.
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration.
- Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

Exemple : les références

```
#include <iostream>

int main (int argc, char **argv)
{
   int i = 10; // i est un entier valant 10
   int &j = i; // j est une référence sur un entier, cet entier est i.
   //int &k = 44; // ligne7 : illégal

   std::cout << "i = " << i << std::endl;
   std::cout << "j = " << j << std::endl;

   // A partir d'ici j est synonyme de i, ainsi :
   j = j + 1; // est équivalent à i = i + 1 !

   std::cout << "i = " << i << std::endl;
   std::cout << "j = " << j << std::endl;

   return 0;
}
```

Exemple : exécution

```
i = 10
j = 10
i = 11
j = 11
```

Si on dé-commente la ligne 7, on obtient cette erreur à la compilation :

```
ligne 7: erreur: invalid initialization of non-const référence of type 'int&'
     from a temporary of type 'int'
```

Intérêt des références

Attention : le = dans la déclaration de la référence n'est pas réellement une affectation puisqu'on ne copie pas la valeur de `i`. En fait, on affirme plutôt le lien entre `i` et `j`. En conséquence, la ligne 7 est donc parfaitement illégal ce que signale le compilateur.

- Comme une référence établit un lien entre deux noms, leur utilisation est efficace dans le cas de variable de grosse taille car cela évitera toute copie.
- Les références sont (systématiquement) utilisées dans le passage des paramètres d'une fonction (ou d'une méthode) dès que le coût d'une recopie par valeur est trop important ("gros" objet).
- Exemple :

```
void truc(const grosObjet& rgo);
```

Références et pointeurs constants (1/2)

L'utilisation des mots clés `const` (et `volatile`) avec les pointeurs et les références est un peu plus compliquée qu'avec les types simples. En effet, il est possible de déclarer des pointeurs sur des variables mais aussi :

- des pointeurs constants sur des variables,
- des pointeurs sur des variables constantes et
- des pointeurs constants sur des variables constantes (idem avec les références).

La position des mots clés `const` (et `volatile`) dans les déclarations des types complexes est donc extrêmement importante.
Remarque : lors de l'analyse de la déclaration d'un identificateur X, il faut toujours commencer par une phrase du type « X est un ... ». Pour trouver la suite de la phrase, il suffit de lire la déclaration en partant de l'identificateur et de suivre l'ordre imposé par les priorités des opérateurs. Cet ordre peut être modifié par la présence de parenthèses.

Références et pointeurs constants (2/2)

Exercice : décrivez les lignes de code suivantes

```
int i = 10;
const int IC = 20;

1) const int *pi; // pi est un pointeur sur un entier constant
-> Laquelle des affectations suivantes est illégale ?
 pic = &IC ; pic = &i ; *pic = IC ; i = *pic ;

2) int const *pi;

3) int j; int * const pi = &j; // pi est un pointeur constant sur un entier non
     constant, que l'on initialise pour référencer la variable j
-> Laquelle des lignes suivantes est légale ?
 a) int * const pi = &j; *pi = i;
 b) int * const pi; pi = &j;
 c) int * const pi = &j; pi = &i ;
```

typdedef

- Le mot réservé `typedef` permet simplement la définition de **synonyme de type** qui peut ensuite être utilisé à la place d'un nom de type :

Exemple d'utilisation de typedef

```
typedef int                 entier;
typedef float               reel;
typedef enum{FALSE,TRUE}    booleen;

entier a; // a de type entier donc de type int
reel   x; // x de type réel donc de type float
```

enum

- enum permet de déclarer un **type énuméré** constitué d'un ensemble de constantes appelées **énumérateurs**.
- Une variable de type énuméré peut recevoir n'importe quel énumérateur (lié à ce type énuméré) comme valeur.
- Le premier énumérateur vaut zéro (par défaut), tandis que tous les suivants correspondent à leur précédent incrémenté de un.

Exemple d'utilisation de enum

```
enum couleur_carte
{
    TREFLE = 1, /* un énumérateur */
    CARREAU, /* 1+1 donc CARREAU = 2 */
    COEUR  = 4, /* en C, les énumérateurs sont équivalents à des entiers (int) */
    PIQUE  = 8 /* il est possible de choisir explicitement les valeurs (ou de
        certaines d'entre elles). */
};
```

Exemple : utilisation des typedef et enum précédents

```
int main() {
   entier e = 1;
   reel r = 2.5;
   booleen fini = FALSE;
   enum couleur_carte carte = CARREAU;
   printf("Le nouveau type entier possède une taille de %d octets (ou %d bits)\n", sizeof(entier), sizeof(
         entier)*8);
   printf("La variable e a pour valeur %d et occupe %d octets\n", e, sizeof(e));
   printf("La variable r a pour valeur %.1f et occupe %d octets\n", r, sizeof(r));
   printf("La variable fini a pour valeur %d et occupe %d octets\n", fini, sizeof(fini));
   printf("La variable carte a pour valeur %d et occupe %d octets\n", carte, sizeof(carte));
   return 0;
}
```

Exemple : exécution

```
Le nouveau type entier possède une taille de 4 octets (ou 32 bits)
La variable e a pour valeur 1 et occupe 4 octets
La variable r a pour valeur 2.5 et occupe 4 octets
La variable fini a pour valeur 0 et occupe 4 octets
La variable carte a pour valeur 2 et occupe 4 octets
```

La pile et le tas

- La mémoire dans un ordinateur est une **succession d'octets (soit 8 bits)**, organisés les uns à la suite des autres et **directement accessibles par une adresse**.
- En C/C++, la mémoire pour stocker des variables est organisée en deux catégories :
  1. la pile (*stack*)
  2. le tas (*heap*)
- Remarque : Dans la plupart des langages de programmation compilés, la pile (*stack*) est l'endroit où sont stockés les paramètres d'appel et les variables locales des fonctions.

La pile (stack)

- La pile (*stack*) est un **espace mémoire réservé au stockage des variables désallouées automatiquement**.
- Sa taille est limitée mais on peut la régler (appel POSIX `setrlimit`).
- La pile est bâtie sur le modèle **LIFO** (*Last In First Out*) ce qui signifie "Dernier Entré Premier Sorti". Il faut voir cet espace mémoire comme une pile d'assiettes où on a le droit d'empiler/dépiler qu'une seule assiette à la fois. Par contre on a le droit d'empiler des assiettes de taille différente. Lorsque l'on ajoute des assiettes on les empile par le haut, les unes au dessus des autres. Quand on les "dépile" on le fait en commençant aussi par le haut, soit par la dernière posée. Lorsqu'une valeur est dépilée elle est effacée de la mémoire.

Le tas (heap)

- Le tas (*heap*) est l'autre **segment de mémoire utilisé lors de l'allocation dynamique** de mémoire durant l'exécution d'un programme informatique.
- Sa taille est souvent considére comme illimitée mais elle est en réalité limitée.
- Les fonctions `malloc` et `free`, ainsi que les opérateurs du langage C++ `new` et `delete` permettent, respectivement, d'allouer et désallouer la mémoire sur le tas.
- La mémoire allouée dans le tas doit être désallouée explicitement.

Sous Linux, on peut visualiser facilement les valeurs du tas et de la pile :

```
$ ulimit -a
...
data seg size           (kbytes, -d) unlimited
...
stack size              (kbytes, -s) 8192
...
```

La taille de la pile étant limitée (ici à 8Mo), cela peut provoquer des écrasements de variables et surtout des "**Erreur de segmentation**" en cas de dépassement. Il est évidemment recommandé d'allouer dans le tas les "grosses" variables sous peine de surprise !

Fuite de mémoire

- L'allocation dynamique dans le tas **ne permet pas la désallocation automatique**.
- Chaque allocation avec "`new`" doit impérativement être libérée (détruite) avec "`delete`" sous peine de créer une **fuite de mémoire**.
- La fuite de mémoire est une zone mémoire qui a été allouée dans le tas par un programme qui a omis de la désallouer avant de se terminer. Cela rend la zone inaccessible à toute application (y compris le système d'exploitation) jusqu'au redémarrage du système. Si ce phénomène se produit trop fréquemment la mémoire se remplit de fuites et le système finit par tomber faute de mémoire.

Ce problème est évité en Java en introduisant le mécanisme de "ramasse-miettes" (*Garbage Collector*).

lvalue

- Une *Left-value* (valeur gauche) est un élément de syntaxe C/C++ pouvant **être écrit à gauche d'un opérateur d'affectation (=)**.
- *Exemple :* une variable, une case de tableau, ...
- Modèle d'une affectation :
  `lvalue = rvalue;` soit `left-value` ← `right-value`
- Une **left-value** doit donc être **un emplacement de stockage en mémoire possédant un type précis**, c'est-à-dire la référence à quelque chose de modifiable.

Affectation d'une lvalue

Exemple :

```
#include <iostream>

int main()
{
   int x;

   x = 5; // affectation de la valeur entière 5 à la variable x (x est une lvalue
        )

   2 = x + 1; // problème car 2 n'est pas une lvalue (2 n'est pas "modifiable") !

   return 0;
}
```

- Le compilateur détecte l'erreur et le message est très clair : "`error: lvalue required as left operand of assignment`"

rvalue

- Une *Right-value* (valeur droite) est un élément de syntaxe C/C++ pouvant **être écrit à droite d'un opérateur d'affectation (=)**.
- *Exemple :* une valeur, une constante, une variable, une expression, ...
- Modèle d'une affectation :
  `lvalue = rvalue;` soit `left-value` ← `right-value`
- Une **right-value** doit être une valeur d'un type précis mais n'a pas forcement de zone de stockage en mémoire.
- Une affectation est encore une **right-value** ... ce qui donne le droit d'écrire : `i = j = 10;`
- C'est équivalent à `i=(j=10);` c'est-à-dire : `j=10; i=j;`

Conversion "forcée" par la lvalue

- Les opérateurs d'affectation (=, -=, += ...), appliqués à des valeurs de type numérique, provoquent la conversion de leur opérande de droite dans le type de leur opérande de gauche. Cette conversion "forcée" peut être "dégradante" (avec perte).

Exemple : transtypage "forcée" avec perte

```
#include <iostream>

int main() {
   int x = 5; float y = 1.5;
   int res;

   res = (x + y); // cela revient à faire int(x + y) ou (int)(x + y) car res est
        de type int
   std::cout << "res = " << res << "\n"; // Affiche : res = 6

   return 0;
}
```

Conversion automatique

- Soit l'opération suivante : `short int a = 2; a + 2;`
- L'opération `a + 2` revient à faire l'addition entre un `short int` (a) et un `int` (2). Cela est impossible car on ne peut réaliser que des opérations entre **type identique**.
- Une conversion implicite (automatique) sera faite (pouvant donner lieu à un *warning* de la part du compilateur).
- Les conversions d'ajustement de type automatique réalisées suivant la hiérarchie ci-dessous sont réalisées **sans perte** :
  1. `char` → `short int` → `int` → `long` → `float` → `double` → `long double`
  2. `unsigned int` → `unsigned long` → `float` → `double` → `long double`

Conversion forcée ou cast

- Une conversion de type (ou de promotion de type) peut être implicite (automatique) ou **explicite** (c'est-à-dire forcée par le programmeur).
- Lorsqu'elle est explicite, on utilise l'**opérateur de *cast*** : `(float)a` permet de forcer le `short int a` en `float` en C.
- Nouvelle syntaxe en C++ : type(expression à transtyper) soit par exemple `float(a)`.
- Les conversions forcées peuvent être des **conversions dégradantes (avec perte)**. Par exemple : `int b = 2.5;`
- En effet, le cast `(int)b` donnera 2 : perte de la partie décimale. Cela peut être dangereux (source d'erreur).

Nouveaux opérateurs de transtypage en C++

- `static_cast` : opérateur de transtypage à tout faire. Ne permet pas de supprimer le caractère `const` ou `volatile`.
- `const_cast` : opérateur spécialisé et limité au traitement des caractères `const` et `volatile`
- `dynamic_cast` : opérateur spécialisé et limité au traitement des *downcast*
- `reinterpret_cast` : opérateur spécialisé dans le traitement des conversions de pointeurs peu portables (permet de réinterpréter les données d'un type en un autre type. Aucune vérification de la validité de cette opération n'est faite).
- La syntaxe est la suivante : `op_cast<expression type>(expression à transtyper)` où `op` prend l'une des valeurs (`static`, `const`, `dynamic` ou `reinterpret`)

L'opérateur static_cast

- C'est l'opérateur de transtypage à tout faire qui remplace dans la plupart des cas l'opérateur hérité du C.
- Toutefois il est limité dans les cas suivants : il ne peut convertir un type constant en type non constant ET il ne peut pas effectuer de promotion (*downcast*).

Exemple : static_cast

```
int i;
double d;

i = static_cast<int>(d);
```

Espace de nom

- En C++, un **espace de nom (*namespace*)** est une notion permettant de lever une ambiguïté sur des termes qui pourraient être homonymes sans cela.
- Il est matérialisé par un préfixe identifiant de manière unique la signification d'un terme. On utilise alors l'opérateur de résolution de portée ::.
- Le terme espace de noms (*namespace*) désigne un lieu abstrait conçu pour accueillir (encapsuler) des ensembles de termes (constantes, variables, ...) appartenant à un même domaine. Au sein d'un même espace de noms, il n'y a pas d'homonymes.
- *Remarque :* la notion d'espace de noms est aussi utilisée en Java, C# et dans les technologies XML.

Utilisation d'un namespace :

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

const int UneConstanteGlobale = 1;
int UneVariableGlobale;

namespace MonEspaceDeNom {
   const int MaConstanteDePorteeNommee = 2;
   int MaVariableDePorteeNommee;
   int UneVariable;
}

int main(int argc, char* argv[]) {
   int UneVariable = 3;
   MonEspaceDeNom::MaVariableDePorteeNommee = UneConstanteGlobale;
   UneVariableGlobale = MonEspaceDeNom::MaConstanteDePorteeNommee;
   MonEspaceDeNom::UneVariable = 4;

   cout << "MonEspaceDeNom::MaVariableDePorteeNommee = " << MonEspaceDeNom::MaVariableDePorteeNommee <<
         endl;
   cout << "UneVariableGlobale = " << UneVariableGlobale << endl;
   cout << "UneVariable = " << UneVariable << endl;
   cout << "MonEspaceDeNom::UneVariable = " << MonEspaceDeNom::UneVariable << endl;
   return 0;
}
```

L'exécution du programme d'essai nous fournit les résultats suivants :

```
MonEspaceDeNom::MaVariableDePorteeNommee = 1

UneVariableGlobale = 2

UneVariable = 3

MonEspaceDeNom::UneVariable = 4
```

Déclaration vs Définition

Avant de poursuivre, il est important de distinguer ce que sont une déclaration et une définition.

- Une **déclaration** s'emploie à énoncer la nature de quelqueChose (son type et son nom par exemple) sans que de la mémoire ne lui soit réservée. Une déclaration énonce donc ce qui va exister.
- Une **définition** désigne l'endroit où a été créée le quelqueChose et où de la mémoire lui a été allouée. Une fois définie, le quelqueChose existe réellement.

Cette distinction existe au niveau des variables, des fonctions, des méthodes, ...

Déclaration vs Définition de fonction

Concernant les fonctions, il faut noter :

- Une **déclaration** de fonction est nécessaire pour la **compilation** (transformation des fichiers sources en fichiers objets)
- Une **définition** de fonction est nécessaire pour l'**édition de liens** (transformation des fichiers objets en un exécutable)

Déclaration et définition d'une fonction

```
// Déclaration d'une fonction de nom foo qui reçoit un entier (int) et qui ne retourne rien (void)
void foo(int); // c'est le prototype de la fonction foo
// noter la présence du point virgule (;)

// Définition de la fonction foo, on précise son contenu (le code)
void foo(int a)
{
   std::cout << "J'ai reçu " << a; // attention : a est une variable locale à la fonction
}
```

Déclaration vs Définition

Remarque :

- Les **déclarations**, qui ne produisent donc aucun code ni aucune allocation mémoire, seront placées dans des **fichiers entêtes** (*header*) d'extension `.h`. Ils seront **pré-compilés**.
- Par opposition, les **définitions**, qui produisent du code et des allocations mémoires, seront placées dans des **fichiers sources** d'extension `.c` (ou `.cpp` ou `.cc` pour le langage C++). Ils seront **compilés**.

- Pour conclure, il faut distinguer :
  - la **déclaration** d'une fonction qui est une instruction fournissant au compilateur un certain nombre d'informations concernant une fonction. Il existe une forme recommandée dite **prototype** :

    `int plus(int, int);` ← fichier en-tête (.h)
  - sa **définition** qui revient à écrire le **corps** de la fonction (qui définit donc les traitements effectués dans le bloc {} de la fonction)

    `int plus(int a, int b) { return a + b; }` ← fichier source (.c ou .cpp)
  - l'**appel** qui est son utilisation. Elle doit correspondre à la déclaration faite au compilateur qui vérifie.

    `int res = plus(2, 2);` ← fichier source (.c ou .cpp)
- *Remarque :* La définition d'une fonction tient lieu de déclaration. Mais elle doit être "connue" avant son utilisation (un appel). Sinon, le compilateur génère un message d'avertissement (warning) qui indiquera qu'il a lui-même fait une déclaration implicite de cette fonction : `"attention : implicit declaration of function 'multiplier'"`

Pointeur vers une fonction

- Le **nom d'une fonction est une constante de type pointeur** :

```
int f(int x, int y)
{
   return x+y;
}

int (*pf)(int, int); // pointeur vers fonction admettant 2 entiers en
    paramètres et retournant un entier

pf = f;           // pointeur vers la fonction f
printf("%d\n", (*pf)(3, 5)); // affiche 8
```

Passage de paramètres

- Lors d'un appel de fonction, les paramètres passés à la fonction doivent correspondre aux paramètres déclarés par leur nombre et par la compatibilité de leurs types.
- En C, il n'existe qu'un seul mode de passage des paramètres : c'est le **passage par valeur**. On dit aussi par **passage par recopie (de valeurs)**. Il faut donc considérer un paramètre comme une variable locale ayant été initialisée avant l'appel de la fonction.

```
int multiplier(int a, int b);

int x = 2; int y = 3; int res;
// Appel de fonction en lui passant les valeurs de x et y
// donc la valeur de x est recopié dans a et la valeur de y est recopié
    dans b
res = multiplier(x, y);
```

Remarque : les noms peuvent être identiques cela ne change rien au mécanisme de recopie !

Problème : tentative de permutation de deux variables

```
#include <stdio.h>
void permuter(int a, int b) {
   int c;
   c = a;
   a = b;
   b = c;
   printf("Dans la fonction après permutation : a = %d et b = %d\n", a,
       b);
}

int main() {
   int a = 2; int b = 3;

   printf("Dans le main : a = %d et b = %d\n", a, b);
   permuter(a, b);
   printf("Après l'appel de la fonction : a = %d et b = %d\n", a, b);

   return 0;
}
```

Mise en évidence du problème de modification de copies

Effectivement, cela ne marche pas car la fonction `permuter` à travailler sur des copies de `a` et `b` :

La permutation ne fonctionne pas :

```
Dans le main : a = 2 et b = 3

Dans la fonction après permutation : a = 3 et b = 2

Après l'appel de la fonction : a = 2 et b = 3
```

Piste : L'exception à cette règle est le cas des **tableaux**. En effet, lorsque le nom d'un tableau constitue l'argument d'une fonction, c'est l'adresse du premier élément qui est transmise. Ses élément ne sont donc pas recopiés.

Permutation d'éléments d'un tableau (1/3)

Ceci reste en effet cohérent avec le fait qu'il n'existe pas de variable désignant un tableau comme un tout. Quand on déclare `int t[10]`, `t` ne désigne pas l'ensemble du tableau. `t` est une **constante de type pointeur** vers un `int` dont la valeur est `&t[0]` (adresse du premier élément du tableau).

```
#include <stdio.h>

void permuter(int t[])
{
   int c;

   c = t[0];
   t[0] = t[1];
   t[1] = c;
   printf("Dans la fonction après permutation : t[0] = %d et t[1] = %d\n
        ", t[0], t[1]);
}
```

Permutation d'éléments d'un tableau (2/3)

Remarque : c'est une très bonne chose en fait car dans le cas d'un "gros tableau", on évite ainsi de recopier toutes les cases. Le **passage d'une adresse** (par exemple 32 bits) sera beaucoup plus efficace et rapide.

```
int main()
{
   int t[2] = { 2, 3 };

   printf("Dans le main : t[0] = %d et t[1] = %d\n", t[0], t[1]);
   permuter(t);
   printf("Après l'appel de la fonction : t[0] = %d et t[1] = %d\n", t
        [0], t[1]);

   return 0;
}
```

Permutation d'éléments d'un tableau (3/3)

Effectivement, cela marche car la fonction `permuter` à travailler avec l'adresse du tableau :

Exemple

```
Dans le main : t[0] = 2 et t[1] = 3

Dans la fonction après permutation : t[0] = 3 et t[1] = 2

Après l'appel de la fonction : t[0] = 3 et t[1] = 2
```

En conséquence, pour réaliser l'effet de **passage de paramètre par adresse**, il suffit alors de **passer en paramètre un pointeur vers la variable**.

Passage par adresse (1/2)

- Pour qu'une fonction puisse modifier le contenu d'une variable passée en paramètre, on doit utiliser en C un **pointeur sur cette variable**.
- Exemple de déclaration : `void permuter(int *pa, int *pb); // pa et pb sont des pointeurs sur des int.`
- Le passage se fait toujours par valeur ou par recopie sauf que maintenant, on passe l'**adresse d'une variable**.

```
void permuter(int *pa, int *pb);
int main() {
   int a = 2; int b = 3;
   printf("Dans le main : a = %d et b = %d\n", a, b);
   permuter(&a, &b); // l'adresse (&) de a est recopiée dans le pointeur
        pa, idem pour l'adresse de b dans pb
   printf("Après l'appel de la fonction : a = %d et b = %d\n", a, b);
   return 0;
}
```

Passage par adresse (2/2)

```
void permuter(int *pa, int *pb) {
   int c;
   c = *pa;
   *pa = *pb;
   *pb = c;
   printf("Dans la fonction après permutation : a = %d et b = %d\n", *pa
        , *pb);
}
```

La permutation fonctionne maintenant :

```
Dans le main : a = 2 et b = 3
Dans la fonction après permutation : a = 3 et b = 2
Après l'appel de la fonction : a = 3 et b = 2
```

Protection contre le passage par adresse (1/4)

Remarque n°1 : lorsqu'on passe une adresse par pointeur, il y a un risque que la fonction modifie cette adresse. Dans ce cas, on peut se protéger en déclarant l'**adresse passée comme constante**.

```
#include <stdio.h>
void foo(int * const pa) {
   int un_autre_a;
   pa = &un_autre_a; // tentative de modification d'adresse contenue
        dans le pointeur
}
int main() { int a = 2; foo(&a); return 0; }
```

On obtient un contrôle d'accès à la compilation :

```
In function 'foo':
erreur: assignment of read-only location 'pa'
```

Protection contre le passage par adresse (2/4)

Remarque n°2 : On a vu que lorsque l'on doit passer en paramètre des "grosses" tailles de variables, il serait plus efficace et plus rapide à l'exécution de passer une adresse. Mais il y aurait un risque que la fonction modifie cette variable. Dans le cas d'un accès limité en lecture, on peut se protéger en déclarant le **pointeur comme constant**.

```
#include <stdio.h>
void foo(const long double *pa) {
   printf("Je suis un pointeur de taille plus légère %d octets\n",
        sizeof(pa));
   printf("Je contiens l'adresse de a : pa = %p\n", pa);
   printf("et j'ai un accès en lecture : a = %.1Lf\n", *pa);
   /* et non, tu ne peux mas modifier le contenu de la variable pointée
        ! */
   printf("mais pas en écriture !\n");
   *pa = 3.5; // ligne 10 à enlever !
}
```

Protection contre le passage par adresse (3/4)

On obtient un contrôle d'accès à la compilation :

```
In function 'foo':
ligne 10: erreur: assignment of read-only location '*pa'
```

Protection contre le passage par adresse (4/4)

```
int main() { long double a = 2.0;
  printf("Je suis une grosse variable de taille %d octets\n",sizeof(a));
  printf("Je suis a et j'ai pour valeur : a = %.1Lf\n", a);
  printf("et pour adresse : &a = %p\n\n", &a);
  foo(&a);
  return 0;
}
```

Une utilisation de pointeur constant :

```
Je suis une grosse variable de taille 12 octets
Je suis a et j'ai pour valeur : a = 2.0
et pour adresse : &a = 0xbf8d8240
Je suis un pointeur de taille plus légère 4 octets
Je contiens l'adresse de a : pa = 0xbf8d8240
et j'ai un accès en lecture : a = 2.0
mais pas en écriture !
```

Passage par référence (1/2)

- En **C++**, on a aussi la possibilité d'utiliser le **passage par référence**.
- Intérêt : lorsqu'on passe des variables (ou des objets) en paramètre de fonctions ou méthodes et que le coût d'une recopie par valeur est trop important ("gros" objet), on choisira un **passage par référence**.

```
void permuter(int &a, int &b) {
 int c;
 c = a; a = b; b = c;
 printf("Dans la fonction après permutation : a = %d et b = %d\n", a,b);
}

int main() {
 int a = 2; int b = 3;
 printf("Dans le main : a = %d et b = %d\n", a, b);
 permuter(a, b);
 printf("Après l'appel de la fonction : a = %d et b = %d\n", a, b);
 return 0;
}
```

Passage par référence (2/2)

La permutation fonctionne également en utilisant un passage par référence :

```
Dans le main : a = 2 et b = 3
Dans la fonction après permutation : a = 3 et b = 2
Après l'appel de la fonction : a = 3 et b = 2
```

Protection contre le passage par référence

Si le paramètre ne doit pas être modifié, on utilisera alors un passage par référence sur une variable constante :

```
void foo(const long double &a) {
   printf("J'ai un accès en lecture : a = %.1Lf\n", a);
   printf("mais pas en écriture !\n");
   //a = 3.5; // interdit car const ! (voir message)
}
int main() { long double a = 2.0;
   printf("Je suis a et j'ai pour valeur : a = %.1Lf\n", a);
   foo(a);
   return 0;
}
```

Le compilateur nous préviendra de toute tentative de modification :

```
In function 'void foo(const long double&)':
erreur: assignment of read-only reference 'a'
```

Bilan

- En résumé, voici les différentes déclarations en fonction du contrôle d'accès désiré sur un paramètre reçu :
  - passage par valeur → accès en lecture seule à la variable passée en paramètre : `void foo(int a);`
  - passage par adresse → accès en lecture seule à la variable passée en paramètre : `void foo(const int *a);`
  - passage par adresse → accès en lecture et en écriture à la variable passée en paramètre : `void foo(int *a);`
  - passage par adresse → accès en lecture et en écriture à la variable passée en paramètre (sans modification de son adresse) : `void foo(int * const a);`
  - passage par adresse → accès en lecture seule à la variable passée en paramètre (sans modification de son adresse) : `void foo(const int * const a);`
  - passage par référence → accès en lecture et en écriture à la variable passée en paramètre : `void foo(int &a);`
  - passage par référence → accès en lecture seule à la variable passée en paramètre : `void foo(const int &a);`

Surcharge

- Il est généralement conseillé d'attribuer des noms distincts à des fonctions différentes.
- Cependant, lorsque des fonctions effectuent la même tache sur des objets de type différent, il peut être pratique de leur attribuer des noms identiques.
- Ce n'est pas possible de réaliser cela en C mais seulement en C++.
- L'utilisation d'un même nom pour des fonctions (ou méthodes) s'appliquant à des types différents est nommée **surcharge**.
- *Remarque :* cette technique est déjà utilisée dans le langage C++ pour les opérateurs de base. L'addition, par exemple, ne possède qu'une seul nom (+) alors qu'il est possible de l'appliquer à des valeurs entières, virgule flottante, etc ...

La surcharge de fonctions en action (1/2)

```
#include <iostream>

using namespace std;

// Un seul nom au lieu de print_int, print_float, ...
void print(int);
void print(float);

int main (int argc, char **argv)
{
   int n = 2;
   float x = 2.5;

   print(n);
   print(x);

   return 0;
}
```

La surcharge de fonctions en action (1/2)

```
void print(int a)
{
   cout << "je suis print et j'affiche un int : " << a << endl;
}

void print(float a)
{
   cout << "je suis print et j'affiche un float : " << a << endl;
}
```

On obtient :

```
je suis print et j'affiche un int : 2
je suis print et j'affiche un float : 2.5
```

Signature et Prototype

- *Remarque importante :* la surcharge ne fonctionne que pour des **signatures différentes** et le type de retour d'une fonction ne fait pas partie de la signature.
- On distingue :
  - La **signature d'une fonction est le nombre et le type de chacun de ses arguments**.
  - Le **prototype d'une fonction est le nombre et le type de ses arguments (signature) et aussi de sa valeur de retour**.
- Aller plus loin : il est aussi possible de **surcharger les opérateurs de base** avec des signatures différentes pour ses propres classes.

Classes et objets

- Les classes sont les éléments de base de la **programmation orientée objet (POO)** en C++. Dans une classe, on réunit :
  - des données variables : les données membres, ou les **attributs** de la classe.
  - des fonctions : les fonctions membres, ou les **méthodes** de la classe.
- Une classe `A` apporte un nouveau type **A** ajouté aux types (pré)définis de base par C++.
- Une variable a créée à partir du type de classe `A` est appelée **instance** (ou **objet**) de la classe A.

Exemple (non compilable*) :

```
class A; // déclare une classe A (* car elle n'est pas définie)

A a1; // instancie un objet a1 de type A
A a2; // crée une instance a2 de type A
```

État et comportement

- Une classe se **déclare dans un fichier `.h`**. La déclaration permet de connaître les types des choses constituant la classe, et ainsi de l'utiliser (directive `#include` pour accéder à sa déclaration).
- Une classe se **définit dans un fichier `.cc` ou `.cpp`**. La définition permettra de fabriquer les choses constituant la classe.
- Un objet possédera une **identité** qui permet de distinguer un objet d'un autre objet (son nom, une adresse mémoire).
- Un objet possédera un **état** (les valeurs contenues dans les attributs propres à cet objet).
- Un objet possédera un **comportement** (l'utilisation de ses méthodes lui fera changer d'état).

La classe string

Ce n'est pas un type de base (c'est une classe) qui permet de stocker (et de manipuler) une suite de lettres (une chaîne de caractères).

```
#include <string>
int main () {
   string le0 = "Chuck"; string le1 = "Norris";
   cout << "Il n'y a que deux éléments en informatique : le " << le0 << " et le
        " << le1 << '.' << endl;
   if(le0 != le1) cout << "Remarque : Les deux chaînes sont différentes" << endl
        ;
   printf("Une chaîne en C : Si windows existe encore c'est parce que %s %s ne s
        'est jamais intéressé à l'informatique.\n", le0.c_str(), le1.c_str());
   return 0;
}
```

Remarque : le "type" `string` remplacera avantageusement le `char *` du langage C en fournissant un type adapté aux chaînes de caractères. L'appel à `c_str()` permettra de récupérer un `char *` utilisable avec des fonctions écrites en C.

Droits d'accès aux membres (1/2)

- Les droits d'accès aux membres d'une classe concernent aussi bien les méthodes que les attributs.
- En C++, on dispose des droits d'accès suivants :
  - Accès `public` : on peut utiliser le membre de n'importe où dans le programme.
  - Accès `private` : seule une fonction membre de la même classe A peut utiliser ce membre ; il est invisible de l'extérieur de A.
  - Accès `protected` : ce membre peut être utilisé par une fonction de cette même classe A, et pas ailleurs dans le programme (ressemble donc à `private`), mais il peut en plus être utilisé par une classe B qui hérite de A.
- Il existe une règle de POO qui précise que **les attributs doivent être encapsulées dans la classe** pour éviter une utilisation extérieure. C'est une forme de protection permettant d'éviter une utilisation incorrecte ou non prévue par le programmeur de la classe. On appliquera ce principe en déclarant l'ensemble des attributs en `private`.

Droits d'accès aux membres (2/2)

- Si c'est nécessaire, l'accès aux attributs privés d'une classe se fera par l'intermédiaire de méthodes nommées des **accesseurs et manipulateurs**. On a l'habitude de les nommer `get()` pour l'accès en lecture et `set()` pour l'accès en écriture d'un attribut.

Exemple (non compilable car les méthodes ne sont pas définies) :

```
class A
{
   private:
      int x; // je suis un attribut privé de type int
      void foo(); // je suis une méthode privée de nom foo

   public: // seulement des méthodes ici (regle POO)
      int getX(); // je suis l'accesseur get de x, je retourne sa valeur
      void setX(int); // je suis le manipulateur set de x, je fixe sa valeur
      void afficher(); // je suis une autre méthode publique
};
```

Constructeur

- Un constructeur est chargé d'**initialiser une instance (objet) de la classe**.
- Il est appelé **automatiquement au moment de la création** de l'objet.
- Un constructeur est une **méthode qui porte toujours le même nom que la classe**.
- Il existe quelques contraintes :
  - Il peut avoir des paramètres, et des valeurs par défaut.
  - Il peut y avoir plusieurs constructeurs pour une même classe.
  - Il n'a jamais de type de retour.
- Il existe d'autres constructeurs : le constructeur par défaut et le constructeur de copie.

Destructeur

- Le destructeur est la **méthode membre appelée lorsqu'une instance (objet) de classe cesse d'exister en mémoire**.
- Son rôle est de **libérer toutes les ressources qui ont été acquises lors de la construction** (typiquement libérer la mémoire qui a été allouée dynamiquement par cet objet).
- Un destructeur est une **méthode qui porte toujours le même nom que la classe, précédé de "~"**.
- Il existe quelques contraintes :
  - Il ne possède aucun paramètre.
  - Il n'y en a qu'un et un seul.
  - Il n'a jamais de type de retour.
- Un destructeur peut être **virtuel**.

Voilà, on sait suffisamment de choses pour écrire notre première classe et instancier nos premiers objets.

Exercice 1

Déclaration d'une classe Point dans un fichier point.h

```
// Une classe apporte un nouveau type : ici un Point. C'est donc un modèle pour les futurs objets de type
      Point.
// Mon type Point possède des caractéristiques (des propriétés) : une coordonnée entière x et une coordonnée
       entière y
// Mon type point peut faire des choses (des opérations) : il est capable d'afficher ses coordonnées, de lui
       demander la valeur de l'une ou l'autre de ses coordonnées et on peut aussi les modifier
      individuellement
class Point
{









};
```

Exercice 2

Définition des membres de la classe Point dans un fichier point.cc

```
#include <iostream> // j'utilise cout
                    // il me faut la déclaration de la classe Point
using namespace std; // utile !

// je definis ici chaque méthode de la classe Point
// pour que le compilateur comprenne et les trouve, j'utilise l'opérateur de résolution de portée ::









// fin du fichier
```

Exercice 3

Un programme d'essai dans un fichier `main.cc`

```
#include <iostream> // j'utilise cout
                    // il me faut la déclaration de la classe Point
using namespace std; // utile !

int main()
{
                    // j'instancie un objet pointA de type Point avec les coordonnées 0,0
                    // ou je crée une instance pointB de type Point avec les coordonnées 1,2

                    // j'appelle la méthode afficher() de l'objet pointA
                    // j'appelle la méthode afficher() de l'objet pointB : chaque objet possède ses propres
                          attributs

                    // je modifie la valeur de l'attribut y de pointB avec la valeur 1
                    // j'appelle la méthode afficher() de l'objet pointB

   return 0;
}
```

Fabrication de l'exécutable a.out (sous Linux)

```
$ g++ -c point.cc
$ g++ -c main.cc
$ ls *.o
main.o point.o
$ g++ main.o point.o

# ou en une seule ligne :
$ g++ main.cc point.cc
```

Exécution du programme d'essai

```
Appel du constructeur Point : j'initialise les membres privés x et y
Appel du constructeur Point : j'initialise les membres privés x et y
0,0
1,2
1,1
Appel du destructeur ~Point : j'ai rien à faire
Appel du destructeur ~Point : j'ai rien à faire
```

new et delete

- Pour **allouer dynamiquement en C++**, on utilisera l'opérateur `new`.
- Celui-ci renvoyant une adresse où est crée la variable en question, il nous faudra un **pointeur** pour la conserver.
- Manipuler ce pointeur, reviendra à manipuler la variable allouée dynamiquement.
- Pour **libérer de la mémoire allouée dynamiquement en C++**, on utilisera l'opérateur `delete`.

Exercice 4

Allocation dynamique d'objet : on utilisera l'opérateur new

```
#include <iostream>
#include "point.h"
using namespace std;

int main()
{
                        // déclaration d'un pointeur pointC sur un objet de type Point

                        // alloue dynamiquement à pointC un objet de type Point avec les coordonnées 3,4

                        // appelle la méthode afficher() : comme pointC est une adresse, on doit utiliser l'
                              opérateur -> pour accéder aux membres de cet objet

                        // modifie la valeur de l'attribut y de pointC avec 5

   (*pointC).afficher(); // cette écriture est possible : on pointe l'objet puis on appelle sa méthode
         afficher()

                        // ne pas oublier de libérer la mémoire allouée pour cet objet

   return 0;
}
```

Exercice 5

Un programme d'essai qui pose problème !

```
#include <iostream>
#include "point.h"
using namespace std;

int main()
{
                        // déclaration d'un objet pointD sans coordonnées (que faut-il faire ?)

                        // déclaration d'un tableauDe10Points contenant 10 objets de type Point

   Point pointE();      // que fait cette ligne ?

   return 0;
}
```

Paramètres par défaut

Le langage C++ offre la possibilité d'avoir des valeurs par défaut pour les paramètres d'une fonction (ou d'une méthode), qui peuvent alors être sous-entendus au moment de l'appel.

```
int mult(int a=2, int b=3) {
   return a*b;
}

mult(3,4); // donne 12
mult(3);   // équivaut à mult(3,3) qui donne 9
mult();    // équivaut à mult(2,3) qui donne 6
```

Remarque : cette possibilité, permet d'écrire qu'un seul constructeur profitant du mécanisme de valeur par défaut.

Constructeur par défaut (1/4)

- Deux constructeurs sont toujours nécessaires dans toute nouvelle classe : le **constructeur par défaut** et le **constructeur de copie**.
- Ils sont tellement importants que si vous ne les écrivez pas, le compilateur tentera de le faire à votre place ... mais à moins de savoir exactement ce que vous faites il vaut mieux les écrire soi-même.
- Le rôle du constructeur par défaut est de créer une instance quand aucun autre constructeur fourni n'est applicable.

Constructeur par défaut (2/4)

Si on essaye de compiler les déclarations suivantes :

```
Point pointD; // un pointD sans coordonnées (que faut-il faire ?)
Point tableauDe10Points[10]; // typiquement : les cases d'un tableau de Point

pointD.afficher();
tableauDe10Points[0].afficher(); // le premier objet Point du tableau
```

On obtient des erreurs en provenance du compilateur :

```
ligne 8: erreur: no matching function for call to Point::Point()'
ligne 9: erreur: no matching function for call to Point::Point()'
```

Constructeur par défaut (3/4)

Le compilateur n'a pas trouvé de constructeur par défaut dans notre classe, il faut donc le déclarer et le définir.

Déclaration d'un constructeur par défaut sans argument :

```
Point();
```

Définition d'un constructeur par défaut sans argument :

```
Point::Point() { x = 0; y = 0; }
```

Maintenant, le programme précédent se compile et le constructeur par défaut sera appelé pour l'objet `pointD` et 10 fois pour chaque objet `Point` du tableau.

Constructeur par défaut (4/4)

Le constructeur par défaut est donc un constructeur **sans argument** ou **avec des arguments qui possèdent une valeur par défaut**.

Déclaration d'un constructeur par défaut avec arguments :

```
Point(int _x=0, int _y=0);
```

Définition d'un constructeur par défaut avec arguments :

```
Point::Point(int _x/*=0*/, int _y/*=0*/) { x = _x; y = _y; }
```

Le constructeur par défaut est nécessaire notamment dans : la création de tableaux d'objet, la fabrication d'objets temporaires et l'agrégation par valeur des objets.

Liste d'initialisation

- Un meilleur moyen d'affecter des valeurs lors de la construction aux données membres de la classe est la **liste d'initialisation**.

Dans le fichier .cc, il suffit d'utiliser la syntaxe suivante :

```
Point::Point(int _x, int _y) : x(_x), y(_y) // c'est la liste d'initialisation
{ /* il n'y a rien d'autre à faire */ }
```

- La liste d'initialisation permet d'utiliser le constructeur de chaque donnée membre, et ainsi d'éviter une affectation après coup.
- la liste d'initialisation doit être utilisée pour certains objets qui ne peuvent pas être contruits par défaut : c'est le cas des références et des objets constants.
- **Important : l'ordre d'initialisation des membres doit correspondre à celui de leurs déclarations dans le fichier .h.**

Le pointeur spécial d'instance : `this`

- Dans certaines conditions particulières, il est nécessaire de disposer d'un moyen de désigner depuis l'intérieur d'une fonction membre, non pas les données membres, mais l'instance elle-même de la classe sur laquelle la méthode membre est appelée.
- Le mot clé **"this" permet de désigner l'adresse de l'instance (de l'objet) sur laquelle la fonction membre a été appelée**.

Exemple d'utilisation du pointeur spécial d'instance this

```
class A {
public: // pour l'exemple
   int x;
   A * getA();
   void setX(int);
};
void A::setX(int x) { this->x = x; } // Ici this permet de distinguer le
     paramètre x de l'attribut x de l'objet de type A sur lequel on a appelé setX

A* A::getA() { return this; } // Ici this est l'adresse de l'objet de type A sur
     lequel on a appelé getA

int main() {
   A a; A *pa;
   a.setX(2);
   pa = a.getA(); // équivalent (très tordu !) à pa = &a !
   cout << "x=" << a.x << endl; // Affiche : x=2
   cout << "x=" << pa->x << endl; // Affiche : x=2
   return 0;
}
```

Agrégation

L'agrégation signifie implicitement « contient un » (ou « est composée d'un » ou « possède un ») ou tout simplement **« a un »**. On distingue deux types d'agrégation :

- Agrégation interne : l'objet agrégé est créé par l'objet agrégateur
- Agrégation externe : l'objet agrégé a été créé extérieurement à l'objet agrégateur

Il y a 3 possibilités de mise en oeuvre :

- agrégation par valeur (appelée aussi **composition**)
- agrégation par référence
- agrégation par pointeur

Remarque : en règle générale, le créateur d'un objet est le responsable de sa destruction.

Agrégation interne par valeur (Composition)

```
class ObjetGraphique
{
   public:
      ObjetGraphique(Point p, int couleur) : point(p), couleur(couleur)
      {}
      void afficher() { cout << point.getX() << "," << point.getY() << " [" <<
           couleur << "]" << endl ; }

   private:
      Point point; // par valeur
      int couleur;
};

Point p1(1, 1);
ObjetGraphique o1(p1, 1);
p1.setX(5);
o1.afficher(); // donne : 1,1 [1]
```

Agrégation externe par référence

```
class ObjetGraphique
{
   public:
      ObjetGraphique(Point &p, int couleur): point(p), couleur(couleur)
      {}
      void afficher() { cout << point.getX() << "," << point.getY() << " [" <<
           couleur << "]" << endl ; }

   private:
      Point &point; // par référence
      int couleur;
};

Point p1(2, 2);
ObjetGraphique o1(p1, 1);
p1.setX(5);
o1.afficher(); // donne : 5,2 [1]
```

Agrégation externe par pointeur

```
class ObjetGraphique {
   public:
      ObjetGraphique(Point *p, int couleur) : point(p), couleur(couleur) {}
      ObjetGraphique(Point &p, int couleur) : point(&p), couleur(couleur) {}
      void afficher() { cout << point->getX() << "," << point->getY() << " [" <<
           couleur << "]" << endl ; }

      private:
         Point *point; // par pointeur
         int couleur;
};

Point p1(3, 3);
Point p2(4, 4);
ObjetGraphique o1(&p1, 1);
ObjetGraphique o2(p2, 1);
p1.setX(5);
p2.setX(6);
o1.afficher(); // donne : 5,3 [1]
o2.afficher(); // donne : 6,4 [1]
```

Constructeur de copie (1/7)

Le constructeur de copie est appelé dans :

- la création d'un objet à partir d'un autre objet pris comme modèle
- le passage en paramètre d'un objet par valeur à une fonction ou une méthode
- le retour d'une fonction ou une méthode renvoyant un objet

La forme habituelle d'un constructeur de copie est la suivante :

```
T(const T&); // T est le nom de la classe
```

Remarque : toute autre duplication (au cours de la vie d'un objet) sera faite par l'opérateur d'affectation (=).

Constructeur de copie (2/7)

Le constructeur de copie est appelé quand on crée un objet à partir d'un autre objet pris comme modèle.

Exemples :

```
Point a; // le constructeur par défaut est appelé

Point b(a); // le constructeur de copie est appelé

Point c = a; // le constructeur de copie est appelé car c'est c n'a pas encore
     été construit

Point d; // le constructeur par défaut est appelé

d = a; // d a déjà été construit donc ici c'est l'opérateur d'affectation qui est
      appelé
```

Constructeur de copie (3/7)

Le constructeur de copie est appelé quand on passe en paramètre un objet par valeur à une fonction ou une méthode et lorsqu'on retourne un objet depuis une fonction ou une méthode.

Combien d'appels au constructeur de copie ici ? 0, 1, 2 ou 3 ?

```
Liste inverser(const Liste l) {
  l.inverser();
  return l;
}

Liste listeA; // ...

Liste listeB = inverser(listeA);
```

Constructeur de copie (4/7)

La forme habituelle d'un constructeur de copie est la suivante :

```
T(const T&);
```

L'objet modèle est passé :

- en référence, ce qui assure de bonnes performances et empêche un bouclage infini
- et la référence est constante, ce qui garantit que seules des méthodes constantes (ne pouvant pas modifier les attributs) seront appelables sur l'objet passé en argument

Constructeur de copie (5/7)

On aura besoin de définir son constructeur de copie :

- Agrégation par valeur : il faut recopier les objets agrégés par valeur en appelant leur constructeur par copie
- Agrégation par référence (ou pointeur) et allocation dynamique de mémoire : la copie optimisée ne générant que des copies de pointeurs, les différents objets utiliseraient les même blocs mémoire.

Exemple d'un constructeur de copie « basique » :

```
T::T(const T& t):a(t.a) { }
```

Remarque : le compilateur fournit un constructeur de copie automatique « par recopie bit à bit optimisée ».

Constructeur de copie (6/7)

Exemple : déclaration d'une classe Ensemble

```
class Ensemble
{
   private:
      enum { capacite_defaut = 100 };
      int * t;
      int   n;
      int   capacite;

   public:
      Ensemble(int capacite=capacite_defaut);
      ~Ensemble();
      Ensemble(const Ensemble &e);
};
```

Constructeur de copie (7/7)

Exemple : définition de la classe Ensemble

```
Ensemble::Ensemble(int capacite/*=capacite_defaut*/):n(0),capacite(capacite)
{
   t = new int[capacite];
   for (int i = 0; i < capacite ; i++) t[i] = 0;
}

Ensemble::~Ensemble()
{
   delete [] t;
}

Ensemble::Ensemble(const Ensemble &e):n(e.n),capacite(e.capacite)
{
   this->t = new int[e.capacite];
   for (int i = 0; i < e.capacite ; i++) this->t[i] = e.t[i];
}
```

Destructeur

On aura besoin de définir son destructeur dès lors que la classe réalise de :

- l'agrégation par pointeur
- l'allocation dynamique de mémoire

Exemple d'un destructeur « basique » :

```
A::A()
{
   t = new int[10];   // allocation dynamique
   point = new Point; // agrégation par pointeur avec allocation dynamique
}

A::~A()
{
   delete [] t;  // on libère
   delete point; // on libère
}
```

Opérateur d'affectation (1/2)

Il est nécessaire de créer un opérateur de copie :

- agrégation
- allocation dynamique de mémoire

Remarque : Dans tous les autres cas, le compilateur crée un opérateur d'affectation « par recopie bit à bit optimisée ».

La forme habituelle d'opérateur d'affectation est la suivante :

```
T& operator=(const T&);
```

Remarque : Cet opérateur renvoie une référence sur T afin de pouvoir l'utiliser avec d'autres affectations.

Opérateur d'affectation (2/2)

Exemple : définition de l'opérateur d'affectation de la classe Ensemble

```
Ensemble & Ensemble::operator = (const Ensemble &e)
{
   if (this != &e)
   {
      delete [] t; // attention l'objet avait déjà été construit
      n = e.n;
      capacite = e.capacite;
      t = new int[e.capacite];
      for (int i = 0; i < e.capacite ; i++) t[i] = e.t[i];
   }
   return *this;
}
```

Rappel : l'opérateur d'affectation est associatif à droite a=b=c est évaluée comme a=(b=c). Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable.

Forme Canonique de Coplien

Une classe T est dite sous forme canonique (ou forme normale ou forme standard) si elle présente les méthodes suivantes :

```
class T
{
   public:
      T (); // Constructeur par défaut
      T (const T&); // Constructeur de copie
      ~T (); // Destructeur éventuellement virtuel
      T &operator= (const T&); // Operateur d'affectation
};
```

Exercice

Décrivez les lignes de code suivantes (mettant en jeu les classes T et U) :

```
1) T t;

2) U u;

3) T z(t);

4) T v();

5) T y = t;

6) z = t;
```

Objets constants

Les règles suivantes s'appliquent aux objets constants :

- On déclare un objet constant avec le modificateur `const`
- On ne peut appliquer que des méthodes constantes sur un objet constant
- Un objet passé en paramètre sous forme de référence constante est considéré comme constant

Remarque : une méthode constante est tout simplement une méthode qui ne modifie aucun des attributs de l'objet. Il est conseillé de qualifier `const` toute fonction qui peut l'être.

Méthodes constantes

La qualification const d'une fonction membre fait partie de sa signature. Ainsi, on peut surcharger une fonction membre non constante par une fonction membre constante :

```
class Point {
   private:
      int x, y;
   public:
      int X() const { return x; }
      int Y() const { return y; }
      int& X() { return x; }
      int& Y() { return y; }
      void afficher() { cout << X() << "," << Y() << endl ; }
};

Point p1(1, 0);
p1.afficher(); // donne : 1,0
int r;
r = p1.X();
p1.Y() = r;
p1.afficher(); // donne : 1,1
```

Attributs constants

Une donnée membre d'une classe peut être qualifiée `const`. Il est alors obligatoire de l'initialiser lors de la construction d'un objet, et sa valeur ne pourra par la suite plus être modifiée.

```
class Point {
   private:
      const char lettre;
      int x, y;

   public:
      Point(char lettre, int x, int y):lettre(lettre), x(x), y(y)
      ...
      void afficher() { cout << lettre << ":" << X() << "," << Y() << endl ; }
};

Point p1('A', 1, 0);
p1.afficher(); // donne : A:1,0
```

Surcharge des opérateurs (1/3)

La surcharge d'opérateur permet aux opérateurs du C++ d'avoir une signification spécifique quand ils sont appliqués à des types spécifiques. Parmi les nombreux exemples que l'on pourrait citer :

- `myString + yourString` pourrait servir à concaténer deux objets `string`
- `maDate++` pourrait servir à incrémenter un objet `Date`
- `a * b` pourrait servir à multiplier deux objets `Nombre`
- `e[i]` pourrait donner accès à un élément contenu dans un objet `Ensemble`

Surcharge des opérateurs (2/3)

Opérateurs du C++ ne pouvant être surchargés :

- `.` : Sélection d'un membre
- `.*` : Appel d'un pointeur de méthode membre
- :: : Sélection de portée
- :? : Opérateur ternaire
- `sizeof, typeid, static_cast, dynamic_cast, const_cast, reinterpret_cast`

Opérateurs C++ qu'il vaut mieux ne pas surcharger :

- `,` : Évaluation séquentielle d'expressions
- `!` : Non logique
- `||` `&&` : Ou et Et logiques

Surcharge des opérateurs (3/3)

Opérateurs C++ que l'on surcharge habituellement :

- Affectation, affectation avec opération (`=`, `+=`, `*=`, etc.) : **Méthode**
- Opérateur « fonction » `()` : **Méthode**
- Opérateur « indirection » `*` : **Méthode**
- Opérateur « crochets » `[]` : **Méthode**
- Incrémentation `++`, décrémentation `--` : **Méthode**
- Opérateur « flèche » et « flèche appel » `->` et `->*` : **Méthode**
- Opérateurs de décalage « et » : **Méthode**
- Opérateurs `new` et `delete` : **Méthode**
- Opérateurs de lecture et écriture sur flux « et » : **Fonction**
- Opérateurs dyadiques genre « arithmétique » (`+`,`-`,`/` etc) : **Fonction**

Surcharge des opérateurs internes

La première technique pour surcharger les opérateurs consiste à les considérer comme des méthodes normales de la classe sur laquelle ils s'appliquent.

```
A Op B se traduit par A.operatorOp(B)

type & operator+=(const type &);

// Dans une classe T :
T & T::operator+=(const T &b) {
   x += b.x;
   return *this;
}
```

Remarque : Les opérateurs de comparaison sont très simples à surcharger. La seule chose essentielle à retenir est qu'ils renvoient une valeur booléenne.

Surcharge des opérateurs externes (1/2)

La deuxième technique utilise la surcharge d'opérateurs externes. La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres.

```
A Op B se traduit par operatorOp(A, B)

// Dans une classe T :
friend T operator+(const T &a, const T &b);

// Et globalement :
T operator+(const T &a, const T &b)
{
   T result = a;
   return result += b;
}
```

Surcharge des opérateurs externes (2/2)

- L'avantage de cette syntaxe est que l'opérateur est réellement symétrique, contrairement à ce qui se passe pour les opérateurs définis à l'intérieur de la classe.
- On constate que les opérateurs externes doivent être déclarés comme étant des **fonctions amies** (`friend`) de la classe sur laquelle ils travaillent, faute de quoi ils ne pourraient pas manipuler les données membres de leurs opérandes.

Opérateurs d'incrémentation et de décrémentation (1/3)

- Les opérateurs d'incrémentation et de décrémentation ont la même notation mais représentent deux opérateurs en réalité. En effet, ils n'ont pas la même signification, selon qu'ils sont placés avant ou après leur opérande. Ne possédant pas de paramètres (ils ne travaillent que sur l'objet), il est donc impossible de les différencier par surcharge.
- La solution qui a été adoptée est de les différencier en donnant un paramètre fictif de type `int` à l'un d'entre eux.
  - opérateurs préfixés : $++$ et $--$ ne prennent pas de paramètre et doivent renvoyer une référence sur l'objet lui-même
  - opérateurs suffixés : $++$ et $--$ prennent un paramètre `int` fictif (que l'on n'utilisera pas) et peuvent se contenter de renvoyer la valeur de l'objet

Opérateurs d'incrémentation et de décrémentation (2/3)

Sous forme de méthode :

```
// Opérateur préfixe :
T &T::operator++(void)
{
   ++x ; // incrémente la variable
   return *this ; // et la retourne
}

// Opérateur suffixe : retourne la valeur et incrémente la variable
T T::operator++(int n)
{
   // crée un objet temporaire,
   T tmp(x); // peut nuire gravement aux performances
   ++x;
   return tmp;
}
```

Opérateurs d'incrémentation et de décrémentation (3/3)

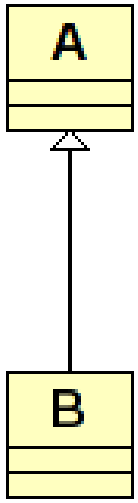
Sous forme de fonctions amies :

```
// Opérateur préfixe
T operator++(T &a)
{
   ++a.x;
   return a;
}

// Opérateur suffixe
T operator++(T &a, int n)
{
   T tmp = a;
   ++a.x;
   return tmp;
}
```

L'héritage (1/2)

- L'héritage (ou spécialisation, ou dérivation) permet d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise.
- L'idée est : "un B **est un** A avec des choses en plus".



Exemple : Un étudiant est une personne, et a donc un nom et un prénom. De plus, il a un numéro INE.

L'héritage (2/2)

```
// A est la classe parente ou mère de B (en anglais superclass)
class A {
   public:
      void f();
};

// B est la classe fille ou dérivée de A : B hérite ou descend de A
class B : public A {
   public:
      void g();
};

A a; B b;

a.f(); // legal : a est un A
b.g(); // legal : b est un B
b.f(); // legal : b est un A (par héritage)
a.g(); // illegal : a n'a pas de membre g() car a n'est pas un B
```

Conversion automatique

Si B hérite de A, alors toutes les instances de B sont aussi des instances de A, et il est donc possible de faire :

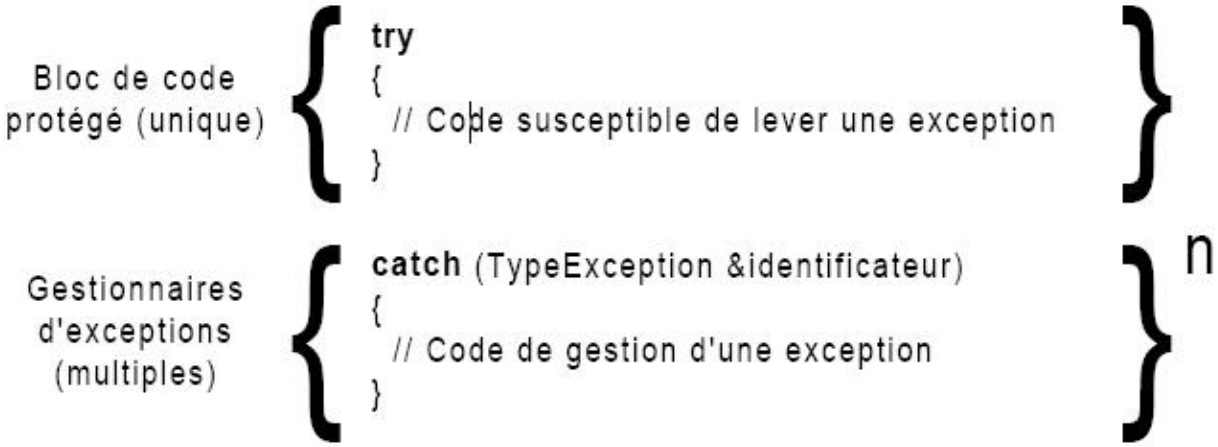
```
A a; B b; a = b;
// Propriété conservée lorsqu'on utilise des pointeurs :
A *pa; B *pb=&b; pa = pb;//car pointer sur un B c'est avant tout pointer sur un A

// Évidement, l'inverse n'est pas vrai :
A a; B b; b = a; // ERREUR !
// Pareil pour les pointeurs :
A *pa=&a; B *pb; pb=pa;//ERREUR : car pointer sur un A n'est pas pointer sur un B
```

Conclusion : Traiter un type dérivé comme s'il était son type de base est appelé transtypage ascendant ou surtypage (*upcasting*). A l'opposé, les transtypage descendant (*downcast*) posent un problème particulier car leur vérification n'est possible qu'à l'exécution. Ils nécessitent l'utilisation d'opérateur de `cast` : `dynamic_cast` (vu plus tard).

Gestion des exceptions (1/2)

Les exceptions ont été rajoutées à la norme du C++ afin de faciliter la mise en oeuvre de code robuste.



Découpage du traitement d'erreur en deux parties :

- le déclenchement : instruction `throw`
- le traitement : deux instructions inséparables `try` et `catch`

Gestion des exceptions (2/2)

Une exception est levée ...

- Si l'instruction en faute n'est pas dans un bloc `try`, il y appel immédiat de la fonction `terminate`.
- Si l'instruction en faute est incluse dans un bloc `try`, le programme saute directement vers les gestionnaires d'exception qu'il examine séquentiellement dans l'ordre du code source :
  - Si l'un des gestionnaires correspond au type de l'exception, il est exécuté, et, s'il ne provoque pas lui même d'interruption ou ne met fin à l'exécution du programme, l'exécution se poursuit à la première ligne de code suivant l'ensemble des gestionnaires d'interruption. En aucun cas il n'est possible de poursuivre l'exécution à la suite de la ligne de code fautive.
  - Si aucun gestionnaire ne correspond au type de l'exception, celle-ci est propagée au niveau supérieur de traitement des exceptions (cas de blocs `try` imbriqués) jusqu'à arriver au programme principal qui lui appellera `terminate`.

(Re)lancer une exception

Exemple :

```
#include <stdexcept> // pour std::range_error
double inverse(double x) {
if(x == 0.0)
      // lance une exception
      throw range_error("Division par zero !\n") ;
else  return 1/x;
}

try
{ inverse(0.0); }
catch (range_error &e)
{
   // traitement local
   throw; // relance l'exception
}
```

Déclarer ses exceptions (1/2)

Selon la norme, les exceptions peuvent être de n'importe quel type (y compris un simple entier). Toutefois, il est utile de les définir en tant que classes. Pour cela, on dérive la classe fournit en standard `std::exception` et on surchargera au minimum la méthode `what()`.

Exemple :

```
class ErreurX: public exception
{
public:
   ErreurX() throw() {}
   ~ErreurX() throw() {}
   const char *what(void) const throw()
   {
      // on peut aussi utiliser un string en privé
      return "Exception sur ...";
   }
};
```

Déclarer ses exceptions (2/2)

Exemple :

```
try
{
   // bloc de code protégé
}
catch (ErreurX &e)
{
   cerr << "Erreur : " << e.what() << endl;
}
catch (exception &e)
{
   cerr << "Exception inconnue : " << e.what() << endl;
   cerr << "Fin du programme" << endl ; // ou pas
   exit(1);
}
```

Les spécificateurs d'exception

Un spécificateur d'exception renseigne l'utilisateur sur le type des exceptions que peut renvoyer une méthode.

```
class A {
   private:
      int x;
   public:
      A(int x = 0) throw (ErreurX);
};
```

Mais, le spécificateur d'exception interdit aux autres méthodes (ou fonctions) appelées d'invoquer des exceptions non prévues. Hors, ce point est difficile à vérifier lors de la compilation. Aussi, les spécificateurs d'exception doivent ils être réservés au code maîtrisé totalement et plus spécifiquement aux méthodes pour lesquelles on est en mesure de prévoir le déroulement complet.