Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://clarolineconnect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Support de cours et livre de référence pour ce cours de C++

- Supports de cours disponibles sur clarolineconnect (pdf sur `http://clarolineconnect.univ-lyon1.fr`).
- Livre *Programmer en langage C++* de Claude Delannoy.

Environnement de Développement Intégré - EDI

Pendant les TDs/TPs, nous utiliserons le compilateur GNU g++ pour le C++ (gcc en C). En particulier, vous pouvez utiliser les EDIs :

- Dev-C++ ;
- **CodeBlocks** ;
- **NetBeans** ;
- eclipse
- Visual Studio 2015 ou 2017.
- Linux/MAC OS (et un makefile ou un projet cmake).

Environnement de Développement Intégré - EDI

Pendant les TDs/TPs, nous utiliserons le compilateur GNU g++ pour le C++ (gcc en C). En particulier, vous pouvez utiliser les EDIs :

- Dev-C++ ;
- **CodeBlocks** ;
- **NetBeans** ;
- eclipse
- Visual Studio 2015 ou 2017.
- Linux/MAC OS (et un makefile ou un projet cmake).

Environnement de Développement Intégré - EDI

Pendant les TDs/TPs, nous utiliserons le compilateur GNU g++ pour le C++ (gcc en C). En particulier, vous pouvez utiliser les EDIs :

- Dev-C++ ;
- **CodeBlocks** ;
- **NetBeans** ;
- eclipse
- Visual Studio 2015 ou 2017.
- Linux/MAC OS (et un makefile ou un projet cmake).

Environnement de Développement Intégré - EDI

Pendant les TDs/TPs, nous utiliserons le compilateur GNU g++ pour le C++ (gcc en C). En particulier, vous pouvez utiliser les EDIs :

- Dev-C++ ;
- **CodeBlocks** ;
- **NetBeans** ;
- eclipse
- Visual Studio 2015 ou 2017.
- Linux/MAC OS (et un makefile ou un projet cmake).

Environnement de Développement Intégré - EDI

Pendant les TDs/TPs, nous utiliserons le compilateur GNU g++ pour le C++ (gcc en C). En particulier, vous pouvez utiliser les EDIs :

- Dev-C++ ;
- **CodeBlocks** ;
- **NetBeans** ;
- eclipse
- Visual Studio 2015 ou 2017.
- Linux/MAC OS (et un makefile ou un projet cmake).

Environnement de Développement Intégré - EDI

Pendant les TDs/TPs, nous utiliserons le compilateur GNU g++ pour le C++ (gcc en C). En particulier, vous pouvez utiliser les EDIs :

- Dev-C++ ;
- **CodeBlocks** ;
- **NetBeans** ;
- eclipse
- Visual Studio 2015 ou 2017.
- Linux/MAC OS (et un makefile ou un projet cmake).

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par ;
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par ;
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par `;`
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par ;
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par ;
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par `;`
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par `;`
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par `;`
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

Est-ce bien le cas ?

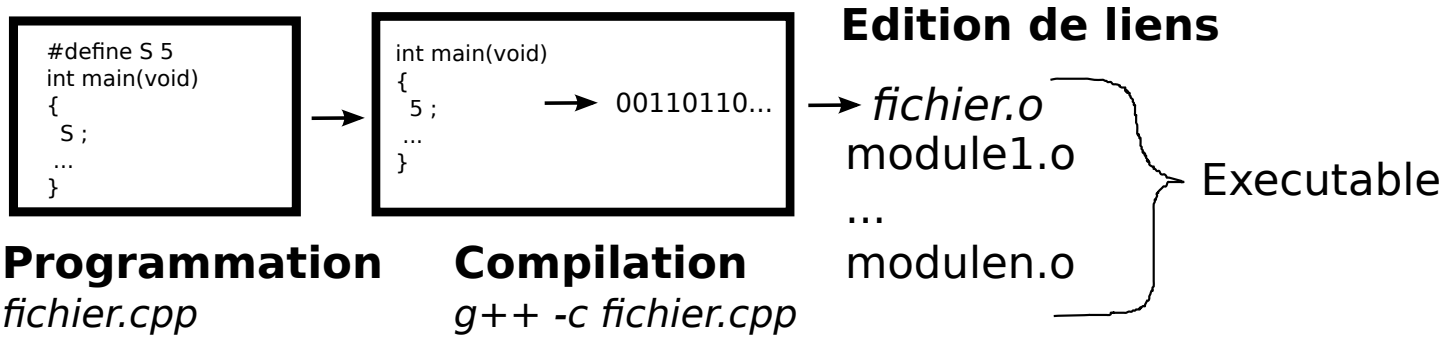
Les connaissances préalables (1/2)

Notions/concepts de la programmation structurée (procédurale) qui doivent être connues :

- Les types scalaires/simples C, les types tableaux, les variables (déclaration, affectation...) et le concept de *lvalues*
- Les expressions C (constantes et variables) : syntaxe, priorité des opérateurs, ordre d'évaluation et évaluation feignante
- Les instructions simples terminées par `;`
- Les structures de contrôle en C (if-else, switch et boucles)
- Les blocs en C { ... }
- Les fonctions C (déclaration/définition) et passages des args
- Les pointeurs C
- Les tableaux 1D et 2D en C et les chaînes de caractères C

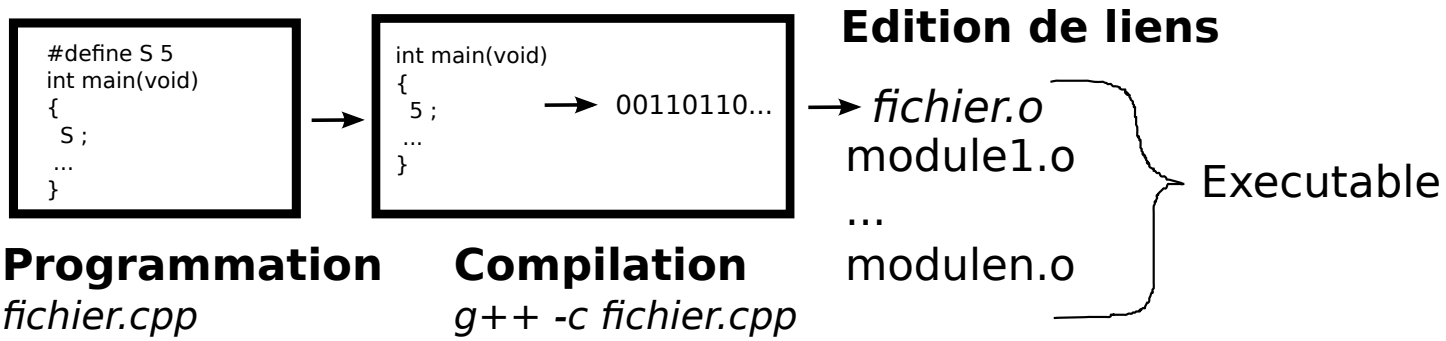
Est-ce bien le cas ?

Les connaissances préalables (2/2)



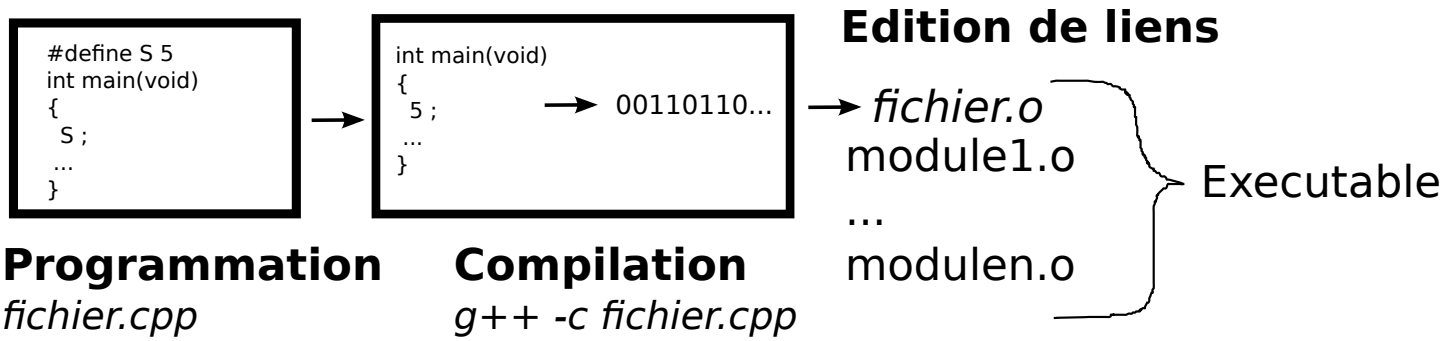
- Les étapes pour transformer un programme dans un langage de haut niveau en un programme bas niveau exécutable par la machine
- Les directives et le préprocesseur
- La notion d'algorithme
- La notion de structure de données (plus générale que la notion de structure C)

Les connaissances préalables (2/2)



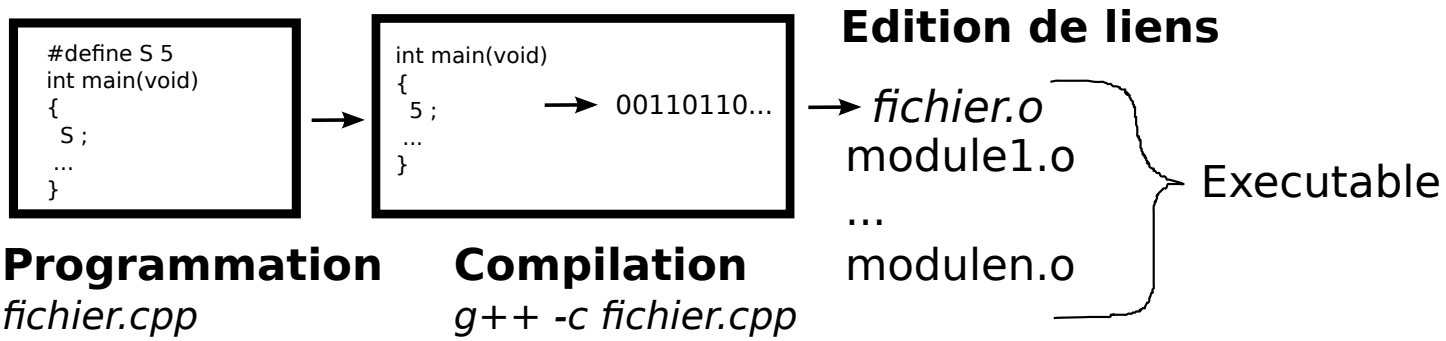
- Les étapes pour transformer un programme dans un langage de haut niveau en un programme bas niveau exécutable par la machine
- Les directives et le préprocesseur
- La notion d'algorithme
- La notion de structure de données (plus générale que la notion de structure C)

Les connaissances préalables (2/2)



- Les étapes pour transformer un programme dans un langage de haut niveau en un programme bas niveau exécutable par la machine
- Les directives et le préprocesseur
- La notion d'algorithme
- La notion de structure de données (plus générale que la notion de structure C)

Les connaissances préalables (2/2)



- Les étapes pour transformer un programme dans un langage de haut niveau en un programme bas niveau exécutable par la machine
- Les directives et le préprocesseur
- La notion d'algorithme
- La notion de structure de données (plus générale que la notion de structure C)

Priorité des opérateurs en C : rappel

1. **référence** () [] -> .
2. **unaire** + − ++ −− ! * & (cast) **sizeof**
3. **arithmétique** * / %
4. **arithmétique** + −
5. **décalage** « »
6. **manipulation de bits** &
7. **manipulation de bits** ∧
8. **manipulation de bits** |
9. **logique** &&
10. **logique** ||
11. **conditionnel** ? :
12. **affectation** = += -= *= /= %= &= ∧= |= «= »=
13. **séquentiel** ,

Objectifs généraux du cours

- **Sensibiliser les étudiants aux purs concepts de la POO** : objet, classe, constructeur, destructeur, héritage, redéfinition, polymorphisme, programmation générique (*templates*)
- **Sensibiliser les étudiants aux aspects spécifiques du C++ ANSI/ISO utiles dans la conception d'application** : références, surdéfinition d'opérateurs, fonctions amies, flots, gestion d'exceptions
- **Développer des compétences attractives pour les entreprises** $\Longrightarrow$ le C et le C++ sont actuellement les langages les plus populaires après le Java (dont la syntaxe est inspirée du C++) !

Objectifs généraux du cours

- **Sensibiliser les étudiants aux purs concepts de la POO** : objet, classe, constructeur, destructeur, héritage, redéfinition, polymorphisme, programmation générique (*templates*)
- **Sensibiliser les étudiants aux aspects spécifiques du C++ ANSI/ISO utiles dans la conception d'application** : références, surdéfinition d'opérateurs, fonctions amies, flots, gestion d'exceptions
- **Développer des compétences attractives pour les entreprises** $\Longrightarrow$ le C et le C++ sont actuellement les langages les plus populaires après le Java (dont la syntaxe est inspirée du C++) !

Objectifs généraux du cours

- **Sensibiliser les étudiants aux purs concepts de la POO** : objet, classe, constructeur, destructeur, héritage, redéfinition, polymorphisme, programmation générique (*templates*)
- **Sensibiliser les étudiants aux aspects spécifiques du C++ ANSI/ISO utiles dans la conception d'application** : références, surdéfinition d'opérateurs, fonctions amies, flots, gestion d'exceptions
- **Développer des compétences attractives pour les entreprises** $\Longrightarrow$ le C et le C++ sont actuellement les langages les plus populaires après le Java (dont la syntaxe est inspirée du C++) !

Objectifs spécifiques du cours

A la fin du semestre, vous devriez être en mesure :

- d'utiliser à bon escient les références C++
- d'implanter une classe "canonique" pour un problème donné
- d'utiliser les conteneurs et la classe string de la bibliothèque standard du C++ (STL)

Et pas seulement !

Objectifs spécifiques du cours

A la fin du semestre, vous devriez être en mesure :

- d'utiliser à bon escient les références C++
- d'implanter une classe "canonique" pour un problème donné
- d'utiliser les conteneurs et la classe string de la bibliothèque standard du C++ (STL)

Et pas seulement !

Objectifs spécifiques du cours

A la fin du semestre, vous devriez être en mesure :

- d'utiliser à bon escient les références C++
- d'implanter une classe "canonique" pour un problème donné
- d'utiliser les conteneurs et la classe string de la bibliothèque standard du C++ (STL)

Et pas seulement !

Objectifs spécifiques du cours

A la fin du semestre, vous devriez être en mesure :

- d'utiliser à bon escient les références C++
- d'implanter une classe "canonique" pour un problème donné
- d'utiliser les conteneurs et la classe string de la bibliothèque standard du C++ (STL)

Et pas seulement !

Plan

Plan

Historique du C++

- **Créé en 1982** par *Bjarne Stroustrup* comme une **extension du C** (inventé en 1972 par Dennis Ritchie)
- **Langage hybride** intégrant le langage C ANSI et un certain nombre de concepts de la POO (Programmation Orientée Objet)
- **Langage en constante évolution**, 4 standards :
  - le **C++98** : la norme C++ publiée par l'ANSI en juillet 1998 qui inclut la bibliothèque standard (*Standard Template Library*)
  - le C++03 : modifications mineures ne concernant pas l'utilisateur
  - le **C++11** (ancient C++0x) de août 2011 qui offre de nouvelles possibilités : utiliser au moins la version 4.3 du compilateur g++
  - le C++17 : std : :any etc.

Historique du C++

- **Créé en 1982** par *Bjarne Stroustrup* comme une **extension du C** (inventé en 1972 par Dennis Ritchie)
- **Langage hybride** intégrant le langage C ANSI et un certain nombre de concepts de la POO (Programmation Orientée Objet)
- **Langage en constante évolution**, 4 standards :
  - le **C++98** : la norme C++ publiée par l'ANSI en juillet 1998 qui inclut la bibliothèque standard (*Standard Template Library*)
  - le C++03 : modifications mineures ne concernant pas l'utilisateur
  - le **C++11** (ancient C++0x) de août 2011 qui offre de nouvelles possibilités : utiliser au moins la version 4.3 du compilateur g++
  - le C++17 : std : :any etc.

Historique du C++

- **Créé en 1982** par *Bjarne Stroustrup* comme une **extension du C** (inventé en 1972 par Dennis Ritchie)
- **Langage hybride** intégrant le langage C ANSI et un certain nombre de concepts de la POO (Programmation Orientée Objet)
- **Langage en constante évolution**, 4 standards :
  - le **C++98** : la norme C++ publiée par l'ANSI en juillet 1998 qui inclut la bibliothèque standard (*Standard Template Library*)
  - le C++03 : modifications mineures ne concernant pas l'utilisateur
  - le **C++11** (ancient C++0x) de août 2011 qui offre de nouvelles possibilités : utiliser au moins la version 4.3 du compilateur g++
  - le C++17 : std : :any etc.

```
#include <iostream> // Entrées et sorties standards : stream signifie flot
#include <cstdlib>  // Pour la définition du symbole EXIT_SUCCESS
#include <cmath>    // Fonctions et symboles mathématiques usuels

using namespace std; // pour pouvoir utiliser les symboles définis
                     // dans l'espace de noms standard std (cout, cin, endl, ...)

int main()   // le programme principal, le point d'entrée du programme
{
   double x; // x est une variable locale à la fonction main
   cout << "Bonjour!" << endl; // flot de sortie << expression

   cout << "Saisissez un nombre positif :";
   cin >> x; // flot d'entrée >> lvalue
   if(x>=0.)
      cout << "La racine carree de " << x << " est " << sqrt(x) << endl;
   else
      cout << "Erreur de saisie!" << endl;

   cout << "Au revoir et a bientot!" << endl;

   return EXIT_SUCCESS;
}
```

Les commentaires /* */ hérités du C et // du C++

/ commentaire sur 1*
*ou plusieurs lignes */*
Ils peuvent apparaître à tout endroit du programme où un espace pourrait être placé, et ils peuvent s'étendre sur plusieurs lignes.

// commentaire sur une seule ligne
Tout ce qui suit le // jusqu'à la fin de ligne est un commentaire.

Déclaration de variables

En C++ :

1. il existe le type **bool** avec 2 valeurs *false* et *true*
2. l'évaluation des expressions booléennes est basée sur le type **bool**, par exemple les opérateurs logiques (&&, ||, !) et relationnels (<, <=, >, >=) renvoient une expression de type **bool** ; en C++ on évite if(x) ..., on préfèrera if(x !=0) ...
3. il existe les références **&** (pointeur géré en interne)
4. **une déclaration d'une variable n'est pas nécessairement en début de bloc**, elle peut se situer après une instruction ; on peut déclarer une variable au sein d'une boucle **for** ; *intérêt ? : éviter d'avoir des variables déclarées non-initialisées*

Déclaration de variables

En C++ :

1. il existe le type **bool** avec 2 valeurs *false* et *true*
2. l'évaluation des expressions booléennes est basée sur le type **bool**, par exemple les opérateurs logiques (&&, ||, !) et relationnels (<, <=, >, >=) renvoient une expression de type **bool** ; en C++ on évite `if(x) ...`, on préfèrera `if(x !=0) ...`
3. il existe les références **&** (pointeur géré en interne)
4. **une déclaration d'une variable n'est pas nécessairement en début de bloc**, elle peut se situer après une instruction ; on peut déclarer une variable au sein d'une boucle **for** ; *intérêt ? : éviter d'avoir des variables déclarées non-initialisées*

Déclaration de variables

En C++ :

1. il existe le type **bool** avec 2 valeurs *false* et *true*
2. l'évaluation des expressions booléennes est basée sur le type **bool**, par exemple les opérateurs logiques (&&, ||, !) et relationnels (<, <=, >, >=) renvoient une expression de type **bool** ; en C++ on évite if(x) ..., on préfèrera if(x !=0) ...
3. il existe les références **&** (pointeur géré en interne)
4. **une déclaration d'une variable n'est pas nécessairement en début de bloc**, elle peut se situer après une instruction ; on peut déclarer une variable au sein d'une boucle **for** ; *intérêt ? : éviter d'avoir des variables déclarées non-initialisées*

Déclaration de variables

En C++ :

1. il existe le type **bool** avec 2 valeurs *false* et *true*
2. l'évaluation des expressions booléennes est basée sur le type **bool**, par exemple les opérateurs logiques (&&, ||, !) et relationnels (<, <=, >, >=) renvoient une expression de type **bool** ; en C++ on évite if(x) ..., on préfèrera if(x !=0) ...
3. il existe les références **&** (pointeur géré en interne)
4. **une déclaration d'une variable n'est pas nécessairement en début de bloc**, elle peut se situer après une instruction ; on peut déclarer une variable au sein d'une boucle **for** ; *intérêt ? : éviter d'avoir des variables déclarées non-initialisées*

Déclaration de variables

En C++ :

1. il existe le type **bool** avec 2 valeurs *false* et *true*
2. l'évaluation des expressions booléennes est basée sur le type **bool**, par exemple les opérateurs logiques (&&, ||, !) et relationnels (<, <=, >, >=) renvoient une expression de type **bool** ; en C++ on évite `if(x) ...`, on préfèrera `if(x !=0) ...`
3. il existe les références **&** (pointeur géré en interne)
4. **une déclaration d'une variable n'est pas nécessairement en début de bloc**, elle peut se situer après une instruction ; on peut déclarer une variable au sein d'une boucle **for** ; *intérêt ? : éviter d'avoir des variables déclarées non-initialisées*

Les expressions constantes calculables par le compilateur

En C :

- utilisation de la directive **#define** TAILLE 100

En C++ :

- **const** *type* ; on peut utiliser une variable constante de type **const int** pour spécifier la taille d'un tableau ou une étiquette d'un **switch**.

Le C++ permet un contrôle sur le type de la constante.

Les expressions constantes calculables par le compilateur

En C :

- utilisation de la directive **#define** TAILLE 100

En C++ :

- **const** *type* ; on peut utiliser une variable constante de type **const int** pour spécifier la taille d'un tableau ou une étiquette d'un **switch**.

Le C++ permet un contrôle sur le type de la constante.

Les directives d'inclusion de fichiers "en-tête" du C

C

- **#include** <stdio.h>
- **#include** <stdlib.h>
- **#include** <math.h>
- **#include** <time.h>
- **#include** <string.h>
- **#include** <limits.h>
- **#include** <float.h>

C++

- **#include** <cstdio>
- **#include** <cstdlib>
- **#include** <cmath>
- **#include** <ctime>
- **#include** <cstring>
- **#include** <climits>
- **#include** <cfloat>

Directives d'inclusion conseillées en C/C++

C

- **#include** <stdio.h>
- **#include** <stdlib.h>
- **#include** <math.h>
- **#include** <time.h>
- **#include** <string.h>
- **#include** <limits.h>
- **#include** <float.h>

C++

- **#include** <iostream>
- **#include** <cstdlib>
- **#include** <cmath>
- **#include** <ctime>
- **#include** <string>
- **#include** <climits>
- **#include** <cfloat>

Opérateur de cast pour convertir dans un type

C

(**newType**) *expression*

C++

- (**newType**) *expression* (style C à éviter)
- const_cast<**newType**> (*expression*)
- static_cast<**newType**> (*expression*)
- reinterpret_cast<**newType**> (*expression*)
- dynamic_cast<**newType**> (*expression*)

Plan

Si T est un type (int, double, un pointeur, ...), **alors T& est le type *référence sur T***. Exemple :

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib>  // Pour la définition du symbole EXIT_SUCCESS

int main() // le programme principal, le point d'entrée du programme
{
   double x = 8.7, y = 2.7;
   double& ref_x = x; // ref_x est une référence sur x (on parle aussi d'alias) ;
                      // il est obligatoire d'initialiser une référence lors de sa création
                      // car les affectations ou in/décrémentations suivantes porteront sur la
                      // variable référencée

   ref_x = 4.2; // toute opération effectuée sur une référence agit sur la variable référencée
                // ici cela revient à faire x = 4.2 ;

   ref_x = ref_x + 1; // ici cela revient à faire x = x + 1 ;

   ref_x = y; // ici cela revient à faire x = y ; (on copie la valeur de y dans x)

   std::cout << ref_x ; // affiche le contenu de la variable x

   return EXIT_SUCCESS;
}
```

Grâce aux références, on peut avoir des paramètres résultats-sortie sans utiliser l'indirection des pointeurs ! En contrepartie, un argument effectif devra être une *lvalue*. Les effets de bords sont cachés à l'appel !

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib>  // Pour la définition du symbole EXIT_SUCCESS

void echange(int&, int&);   // déclaration de echange

int main() // le programme principal, le point d'entrée du programme
{
   int n = 7, m = 4;
   std::cout << " Avant echange, n = " << n << " et m = " << m << std::endl;
   echange(n, m); // grace aux références, plus besoin de passer directement l'adresse
                  // le passage des adresses reste caché à l'utilisateur final
   std::cout << " Apres echange, n = " << n << " et m = " << m << std::endl;

   return EXIT_SUCCESS;
}

void echange(int& a, int& b) // définition de echange
{
   int tmp = a;
   a = b;
   b=tmp;
}
```

Attention, il ne sera pas possible de modifier une constante ou une *lvalue* d'un autre type.

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib>  // Pour la définition du symbole EXIT_SUCCESS

void modifie(double&);   // déclaration de modifie

int main() // le programme principal, le point d'entrée du programme
{
   double x =1.3;
   modifie(x);   // OK
   modifie(5.2); // ERREUR
   const double pi = 3.14159;
   modifie(pi);  // ERREUR
   int i=5;
   modifie(i);   // ERREUR

   return EXIT_SUCCESS;
}

void modifie(double& d) // définition de modifie
{
   d *= 1.2;
}
```

Cela va permettre principalement d'appeler une fonction retournant une référence (sur une *lvalue* non locale) et de l'utiliser comme une *lvalue* (affectation, incrémentation...).

```
#include <cstdlib> // Pour la définition du symbole EXIT_SUCCESS

int erreur=-1;
int t[100];
int& tableau(int);       // déclaration de tableau

int main() // le programme principal, le point d'entrée du programme
{
   for(int i = 0; i<100; i++)
      tableau(i) = 5;

   for(int i = 0; i<100; i++)
      tableau(i)++;

   return EXIT_SUCCESS;
}

int& tableau(int indice) // définition de tableau
{
   if( indice>=0 && indice<100 ) // modification sécurisée
      return t[indice];

   return erreur;
}
```

Pour des arguments données ou entrée d'une fonction, il peut être souhaitable d'utiliser les références si on ne souhaite pas recopier tous les champs d'une *lvalue* structurée. Dans ce cas pour interdire la modification de la lvalue, il faut précéder le type référencé du mot clef **const**.

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib> // Pour la définition du symbole EXIT_SUCCESS

void affiche(const MonType& arg) // définition de affiche sans déclaration
{
   // ici on peut seulment accéder au contenu de arg, mais on ne peut pas le modifier
   // si une instruction essaie de faire une modification, le code ne compilera pas
   ...
}

int main() // le programme principal, le point d'entrée du programme
{
   MonType t;

   t = ... ;

   affiche(t) ;

   return EXIT_SUCCESS;
}
```

L'utilisation d'une référence constante est assez subtile ; elle autorise les conversions implicites.

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib>  // Pour la définition du symbole EXIT_SUCCESS

void ne_modifie_pas(const double&);   // déclaration de ne_modifie_pas

int main() // le programme principal, le point d'entrée du programme
{
   double x =1.3;
   ne_modifie_pas(x);   // OK
   ne_modifie_pas(5.2); // OK : création d'une copie locale
   const double pi = 3.14159;
   ne_modifie_pas(pi);  // OK
   const int i=5;
   ne_modifie_pas(i);   // OK : création d'une copie locale dans laquelle la valeur convertie
                        // en double est stockée
   int j=5;
   ne_modifie_pas(j);   // OK : création d'une copie locale dans laquelle la valeur convertie
                        // en double est stockée
   return EXIT_SUCCESS;
}

void ne_modifie_pas(const double& d) // définition de ne_modifie_pas
{
   std::cout << d << " n'est pas modifie!" << std::endl;
}
```

Plan

Le C++ autorise la surcharge

Surcharge

Possibilité offerte qui permet d'utiliser des fonctions avec le **même identificateur**, le même type de retour, **mais pas les mêmes arguments formels**.

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib>  // Pour la définition du symbole EXIT_SUCCESS
float somme(float a, float b)
{
   return a+b;
}
float somme(float a, float b, float c)
{
   return a+b+c;
}
int main() // le programme principal, le point d'entrée du programme
{
   cout << " 5 + 7 = " << somme(5,7) << " et 5 + 7 + 9 = " << somme(5,7,9) << endl;
   return EXIT_SUCCESS;
}
```

Ambiguités de la surcharge

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib> // Pour la définition du symbole EXIT_SUCCESS

void truc(int a);
void truc(const int a);   // ERREUR : pas possible de distinguer
void chose(int &);
void chose(int);          // ERREUR : pas possible de distinguer
void machin(int &);
void machin(const int &); // OK : l'attribut const est différentiateur
                          // pour des références
double somme(int a, double b)
{
   return a+b;
}
double somme(double a, int b)
{
   return a+b;
}
int main() // le programme principal, le point d'entrée du programme
{
   somme(7,5); // ERREUR DE COMPILATION : AMBIGUITE NON RESOLUE
   return EXIT_SUCCESS;
}
```

Fonction inline

Le compilateur **remplace l'appel à la fonction inline par le code de cette dernière, les arguments effectifs remplaçant les arguments formels**.
L'objectif est un gain d'efficacité : on gagne le temps nécéssaire pour empiler et dépiler les paramètres de la fonction, pour retourner une valeur ainsi que pour sauter vers le code de la fonction.

```
#include <iostream> // Entrées et sorties standards
#include <cstdlib> // Pour la définition du symbole EXIT_SUCCESS
inline double abs(double x)
{
   return x<0 ? -x : x ;
}
int main() // le programme principal, le point d'entrée du programme
{
   cout << abs(-8.3) << endl;
   return EXIT_SUCCESS;
}
```

Quand doit-on déclarer une fonction inline ? Seulement si ces trois conditions sont vérifiées :

- si la fonction est fréquemment appelée.
- si elle est de petite taille (sinon le code va devenir démesurément volumineux)
- si les traitements effectués par la fonction sont rapides (si on a un traitement de plusieurs secondes par appel alors le gain est négligeable)

Les fonctions inline permettent de remplacer les macros du C tout en pouvant effectuer les contrôles syntaxiques et sémantiques des fonctions usuelles.

Quand doit-on déclarer une fonction inline ? Seulement si ces trois conditions sont vérifiées :

- si la fonction est fréquemment appelée.
- si elle est de petite taille (sinon le code va devenir démesurément volumineux)
- si les traitements effectués par la fonction sont rapides (si on a un traitement de plusieurs secondes par appel alors le gain est négligeable)

Les fonctions inline permettent de remplacer les macros du C tout en pouvant effectuer les contrôles syntaxiques et sémantiques des fonctions usuelles.

Quand doit-on déclarer une fonction inline ? Seulement si ces trois conditions sont vérifiées :

- si la fonction est fréquemment appelée.
- si elle est de petite taille (sinon le code va devenir démesurément volumineux)
- si les traitements effectués par la fonction sont rapides (si on a un traitement de plusieurs secondes par appel alors le gain est négligeable)

Les fonctions inline permettent de remplacer les macros du C tout en pouvant effectuer les contrôles syntaxiques et sémantiques des fonctions usuelles.

Quand doit-on déclarer une fonction inline ? Seulement si ces trois conditions sont vérifiées :

- si la fonction est fréquemment appelée.
- si elle est de petite taille (sinon le code va devenir démesurément volumineux)
- si les traitements effectués par la fonction sont rapides (si on a un traitement de plusieurs secondes par appel alors le gain est négligeable)

Les fonctions inline permettent de remplacer les macros du C tout en pouvant effectuer les contrôles syntaxiques et sémantiques des fonctions usuelles.

En C++, à la *déclaration* d'une fonction de m arguments, il est possible de donner une valeur par défaut au $(m-n)^{eme}$ argument, seulement si les n derniers arguments ont tous une valeur par défaut.

- void machin(int a, double c, bool cas_simple=true) ;
- void machin2(int a, double c=5.2, bool cas_simple=true) ;
- void machin3(int a=8, double c=5.2, bool cas_simple=true) ;

Lors de l'appel à une fonction, *les paramètres effectifs ne sont pas obligatoires pour les arguments formels avec une valeur par défaut*. S'ils sont omis, ça sera la valeur par défaut qui sera utilisée.

```
machin(4, 2.7) ; machin2(4) ; machin3() ; // appels valides
machin(4) ; machin2() ; // appels non valides
```

En C++, à la *déclaration* d'une fonction de m arguments, il est possible de donner une valeur par défaut au $(m-n)^{eme}$ argument, seulement si les n derniers arguments ont tous une valeur par défaut.

- void machin(int a, double c, bool cas_simple=true) ;
- void machin2(int a, double c=5.2, bool cas_simple=true) ;
- void machin3(int a=8, double c=5.2, bool cas_simple=true) ;

Lors de l'appel à une fonction, *les paramètres effectifs ne sont pas obligatoires pour les arguments formels avec une valeur par défaut*. S'ils sont omis, ça sera la valeur par défaut qui sera utilisée.

```
machin(4, 2.7) ; machin2(4) ; machin3() ; // appels valides
machin(4) ; machin2() ; // appels non valides
```

Plan

Les flots d'entrée-sortie de C++ : cin, cout, cerr et clog

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

- cin associé à l'entrée standard : lecture au clavier ;
- cout associé à la sortie standard : affichage à l'écran ;
- cerr associé à la sortie erreur standard ;
- clog associé à la sortie erreur standard, mais les données sont bufférisées.

Les flots d'entrée-sortie de C++ : cin, cout, cerr et clog

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

- cin associé à l'entrée standard : lecture au clavier ;
- cout associé à la sortie standard : affichage à l'écran ;
- cerr associé à la sortie erreur standard ;
- clog associé à la sortie erreur standard, mais les données sont bufférisées.

Les flots d'entrée-sortie de C++ : cin, cout, cerr et clog

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

- cin associé à l'entrée standard : lecture au clavier ;
- cout associé à la sortie standard : affichage à l'écran ;
- cerr associé à la sortie erreur standard ;
- clog associé à la sortie erreur standard, mais les données sont bufférisées.

Les flots d'entrée-sortie de C++ : cin, cout, cerr et clog

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

- cin associé à l'entrée standard : lecture au clavier ;
- cout associé à la sortie standard : affichage à l'écran ;
- cerr associé à la sortie erreur standard ;
- clog associé à la sortie erreur standard, mais les données sont bufférisées.

Pourquoi utilise-t-on un espace de nom appelé std ?

- Pour éviter d'utiliser des fonctions ou des symboles définis dans un autre module ou une autre bibliothèque : des définitions sont "rangées" dans un espace de noms (**namespace**). Cela ressemble aux packages du Java.
- Pour accéder à la variable *V*, à la fonction *fonc* ou au symbole S définis dans l'espace de noms *maLib*, il faut faire *maLib: :V*, *maLib: :fonc* ou *maLib: :S*.
- Pour éviter d'écrire *maLib : :*, on peut utiliser l'instruction *using* : **using namespace** maLib ; au début du fichier.

Pourquoi utilise-t-on un espace de nom appelé std ?

- Pour éviter d'utiliser des fonctions ou des symboles définis dans un autre module ou une autre bibliothèque : des définitions sont "rangées" dans un espace de noms (**namespace**).Cela ressemble aux packages du Java.
- Pour accéder à la variable *V*, à la fonction *fonc* ou au symbole S définis dans l'espace de noms *maLib*, il faut faire *maLib: :V*, *maLib: :fonc* ou *maLib: :S*.
- Pour éviter d'écrire *maLib : :*, on peut utiliser l'instruction *using* : **using namespace** maLib ; au début du fichier.

Pourquoi utilise-t-on un espace de nom appelé std ?

- Pour éviter d'utiliser des fonctions ou des symboles définis dans un autre module ou une autre bibliothèque : des définitions sont "rangées" dans un espace de noms (**namespace**).Cela ressemble aux packages du Java.
- Pour accéder à la variable *V*, à la fonction *fonc* ou au symbole S définis dans l'espace de noms *maLib*, il faut faire *maLib: :V*, *maLib: :fonc* ou *maLib: :S*.
- Pour éviter d'écrire *maLib : :*, on peut utiliser l'instruction *using* : **using namespace** maLib ; au début du fichier.

Pourquoi utilise-t-on un espace de nom appelé std ?

- Pour éviter d'utiliser des fonctions ou des symboles définis dans un autre module ou une autre bibliothèque : des définitions sont "rangées" dans un espace de noms (**namespace**).Cela ressemble aux packages du Java.
- Pour accéder à la variable *V*, à la fonction *fonc* ou au symbole S définis dans l'espace de noms *maLib*, il faut faire *maLib: :V*, *maLib: :fonc* ou *maLib: :S*.
- Pour éviter d'écrire *maLib : :*, on peut utiliser l'instruction *using* : **using namespace** maLib ; au début du fichier.

Analysons la ligne de code suivante : `cout « "Mon texte" ;`

Analysons la ligne de code suivante : `cout « 5+9 ;`

`cout « 5+9` est équivalent à `operateur«(cout, 5+9)`

« est un **opérateur** (fonction) qui a deux opérandes :

- un flot de sortie (`cout`) comme opérande de gauche ;
- une expression (chaîne, caractère, entier, flottant) comme opérande de droite.

Analysons la ligne de code suivante : `cout « "Mon texte" ;`
Analysons la ligne de code suivante : `cout « 5+9 ;`

`cout « 5+9` est équivalent à `operateur«(cout, 5+9)`

« est un **opérateur** (fonction) qui a deux opérandes :

- un flot de sortie (`cout`) comme opérande de gauche ;
- une expression (chaîne, caractère, entier, flottant) comme opérande de droite.

Analysons la ligne de code suivante : `cout « "Mon texte" ;`
Analysons la ligne de code suivante : `cout « 5+9 ;`

`cout « 5+9` est équivalent à operateur«(cout, 5+9)

« est un **opérateur** (fonction) qui a deux opérandes :

- un flot de sortie (cout) comme opérande de gauche ;
- une expression (chaîne, caractère, entier, flottant) comme opérande de droite.

Analysons la ligne de code suivante : `cout « "Mon texte" ;`
Analysons la ligne de code suivante : `cout « 5+9 ;`

`cout « 5+9` est équivalent à operateur«(cout, 5+9)

« est un **opérateur** (fonction) qui a deux opérandes :

- un flot de sortie (`cout`) comme opérande de gauche ;
- une expression (chaîne, caractère, entier, flottant) comme opérande de droite.

Un opérateur « est défini pour chaque type scalaire (explicitement ou grace à une convertion vers un type pour lequel l'opérateur existe) ! Le compilateur choisit le bon grace au type.

- ostream& **operator**«(ostream&, char) ;
- ostream& **operator**«(ostream&, int) ;
- ostream& **operator**«(ostream&, double) ;
- ostream& **operator**«(ostream&, const char *) ;
- ...

c'est la **surdéfinition/surcharge d'opérateurs** !

Grace à elle, plus besoin de code format comme avec printf !

Un opérateur « est défini pour chaque type scalaire (explicitement ou grace à une convertion vers un type pour lequel l'opérateur existe) ! Le compilateur choisit le bon grace au type.

- ostream& **operator**«(ostream&, char) ;
- ostream& **operator**«(ostream&, int) ;
- ostream& **operator**«(ostream&, double) ;
- ostream& **operator**«(ostream&, const char *) ;
- ...

c'est la **surdéfinition/surcharge d'opérateurs** !

Grace à elle, plus besoin de code format comme avec printf !

Un opérateur « est défini pour chaque type scalaire (explicitement ou grace à une convertion vers un type pour lequel l'opérateur existe) ! Le compilateur choisit le bon grace au type.

- ostream& **operator**«(ostream&, char) ;
- ostream& **operator**«(ostream&, int) ;
- ostream& **operator**«(ostream&, double) ;
- ostream& **operator**«(ostream&, const char *) ;
- ...

c'est la **surdéfinition/surcharge d'opérateurs** !

Grace à elle, plus besoin de code format comme avec printf !

L'opérateur « est associatif de gauche à droite, ainsi
cout « "La valeur est : " « 5+9 ; est équivalent à
(cout « "La valeur est : ") « 5+9 ;

on peut donc "sérialiser" n cout « expression_i ; sur une seule ligne :
cout « expression_1 « expression_2 « ... « expression_n ;

L'opérateur « est associatif de gauche à droite, ainsi
cout « "La valeur est : " « 5+9 ; est équivalent à
(cout « "La valeur est : ") « 5+9 ;

on peut donc "sérialiser" n cout « expression_i ; sur une seule ligne :
cout « expression_1 « expression_2 « ... « expression_n ;

Analysons la ligne de code suivante : `int n ; cin » n ;`

`cin » n` est équivalent à operateur»(cin, n)

» est un **opérateur** (fonction) qui a deux opérandes :

- un flot d'entrée (`cin`) comme opérande de gauche ;
- une *lvalue* (zone mémoire de type chaîne, caractère, entier, flottant) comme opérande de droite.

Attention : pour lire une variable de type `bool`, seules les entrées claviers 0 et 1 sont valides.

Analysons la ligne de code suivante : `int n ; cin » n ;`
`cin » n` est équivalent à operateur»(cin, n)

» est un **opérateur** (fonction) qui a deux opérandes :

- un flot d'entrée (`cin`) comme opérande de gauche ;
- une *lvalue* (zone mémoire de type chaîne, caractère, entier, flottant) comme opérande de droite.

Attention : pour lire une variable de type `bool`, seules les entrées claviers 0 et 1 sont valides.

Analysons la ligne de code suivante : `int n ; cin » n ;`
`cin » n` est équivalent à operateur»(cin, n)

» est un **opérateur** (fonction) qui a deux opérandes :

- un flot d'entrée (`cin`) comme opérande de gauche ;
- une *lvalue* (zone mémoire de type chaîne, caractère, entier, flottant) comme opérande de droite.

Attention : pour lire une variable de type `bool`, seules les entrées claviers 0 et 1 sont valides.

Analysons la ligne de code suivante : `int n ; cin » n ;`
`cin » n` est équivalent à operateur»(cin, n)

» est un **opérateur** (fonction) qui a deux opérandes :

- un flot d'entrée (`cin`) comme opérande de gauche ;
- une *lvalue* (zone mémoire de type chaîne, caractère, entier, flottant) comme opérande de droite.

Attention : pour lire une variable de type **bool**, seules les entrées claviers 0 et 1 sont valides.

Un opérateur » est défini pour chaque type scalaire.

- istream& **operator**»(istream&, char&) ;
- istream& **operator**»(istream&, int&) ;
- istream& **operator**»(istream&, double&) ;
- istream& **operator**»(istream&, char *) ;
- ...

Grace à la surcharge d'opérateurs, plus besoin de code format comme avec scanf !

Un opérateur » est défini pour chaque type scalaire.

- istream& **operator**»(istream&, char&) ;
- istream& **operator**»(istream&, int&) ;
- istream& **operator**»(istream&, double&) ;
- istream& **operator**»(istream&, char *) ;
- ...

Grace à la surcharge d'opérateurs, plus besoin de code format comme avec scanf !

L'opérateur » est associatif de gauche à droite, ainsi
`int n, m ; cin » n » m ;` est équivalent à
`(cin » n) » m ;`

on peut donc "sérialiser" n `cin » lvalue_i ;` sur une seule ligne :
`cin » lvalue_1 » lvalue_2 » ... » lvalue_n ;`

L'opérateur » est associatif de gauche à droite, ainsi
`int n, m ; cin » n » m ;` est équivalent à
`(cin » n) » m ;`

on peut donc "sérialiser" n `cin » lvalue_i ;` sur une seule ligne :
`cin » lvalue_1 » lvalue_2 » ... » lvalue_n ;`

Précisions sur le fonctionnement de cin

- Par défaut, toute lecture commence par avancer le pointeur/la tête de lecture jusqu'au 1er caractère non-séparateur (les 2 principaux séparateurs sont l'espace et la fin de ligne \n).
- Et la lecture continue jusqu'à la présence d'un séparateur, d'un caractère invalide (une lettre pour un nombre, un '.' pour un entier...) ou la lecture d'1 caractère pour une variable de type `char`.
- **Les caractères non-exploités restent dans le tampon de lecture pour les prochaines lectures.**

Attention : si la lecture d'un caractère invalide survient dès le 1er caractère lu, alors le flot `cin` est bloqué (la tête de lecture n'avance plus pour les lectures suivantes).

Précisions sur le fonctionnement de cin

- Par défaut, toute lecture commence par avancer le pointeur/la tête de lecture jusqu'au 1er caractère non-séparateur (les 2 principaux séparateurs sont l'espace et la fin de ligne \n).
- Et la lecture continue jusqu'à la présence d'un séparateur, d'un caractère invalide (une lettre pour un nombre, un '.' pour un entier...) ou la lecture d'1 caractère pour une variable de type **char**.
- **Les caractères non-exploités restent dans le tampon de lecture pour les prochaines lectures.**

Attention : si la lecture d'un caractère invalide survient dès le 1er caractère lu, alors le flot cin est bloqué (la tête de lecture n'avance plus pour les lectures suivantes).

Précisions sur le fonctionnement de cin

- Par défaut, toute lecture commence par avancer le pointeur/la tête de lecture jusqu'au 1er caractère non-séparateur (les 2 principaux séparateurs sont l'espace et la fin de ligne \n).
- Et la lecture continue jusqu'à la présence d'un séparateur, d'un caractère invalide (une lettre pour un nombre, un '.' pour un entier...) ou la lecture d'1 caractère pour une variable de type **char**.
- **Les caractères non-exploités restent dans le tampon de lecture pour les prochaines lectures**.

Attention : si la lecture d'un caractère invalide survient dès le 1er caractère lu, alors le flot cin est bloqué (la tête de lecture n'avance plus pour les lectures suivantes).

Précisions sur le fonctionnement de cin

- Par défaut, toute lecture commence par avancer le pointeur/la tête de lecture jusqu'au 1er caractère non-séparateur (les 2 principaux séparateurs sont l'espace et la fin de ligne \n).
- Et la lecture continue jusqu'à la présence d'un séparateur, d'un caractère invalide (une lettre pour un nombre, un '.' pour un entier...) ou la lecture d'1 caractère pour une variable de type **char**.
- **Les caractères non-exploités restent dans le tampon de lecture pour les prochaines lectures**.

Attention : si la lecture d'un caractère invalide survient dès le 1er caractère lu, alors le flot cin est bloqué (la tête de lecture n'avance plus pour les lectures suivantes).

Formatages à l'aide de manipulateurs (1/2)

Pour avoir accès à tous les formatages :

- **#include** <iomanip>
- **un manipulateur s'insère dans le flot comme une donnée, et il s'applique à partir de son insertion**

Formatages du flot de sortie *cout* :

- endl : insère une fin de ligne.
- setw(x) : prochaine sortie sur x caractères minimum quitte à rajouter des espaces.
- setfill(c) : utilise c comme caractère de remplissage.
- setprecision(n) : fixe à n le nombre de chiffres d'affichage.

```
cout « setw(2) « setfill('0') « 2 « endl;
```

Formatages à l'aide de manipulateurs (1/2)

Pour avoir accès à tous les formatages :

- **#include** <iomanip>
- **un manipulateur s'insère dans le flot comme une donnée, et il s'applique à partir de son insertion**

Formatages du flot de sortie *cout* :

- endl : insère une fin de ligne.
- setw(x) : prochaine sortie sur x caractères minimum quitte à rajouter des espaces.
- setfill(c) : utilise c comme caractère de remplissage.
- setprecision(n) : fixe à n le nombre de chiffres d'affichage.

```
cout « setw(2) « setfill('0') « 2 « endl ;
```

Formatages à l'aide de manipulateurs (1/2)

Pour avoir accès à tous les formatages :

- **#include** <iomanip>
- **un manipulateur s'insère dans le flot comme une donnée, et il s'applique à partir de son insertion**

Formatages du flot de sortie *cout* :

- endl : insère une fin de ligne.
- setw(x) : prochaine sortie sur x caractères minimum quitte à rajouter des espaces.
- setfill(c) : utilise c comme caractère de remplissage.
- setprecision(n) : fixe à n le nombre de chiffres d'affichage.

```
cout « setw(2) « setfill('0') « 2 « endl;
```

Formatages à l'aide de manipulateurs (2/2)

Formatages du flot d'entrée *cin* :

- setw(x) : lire x caractères au maximum pour la prochaine chaîne de style C (si plusieurs chaînes de style C, alors il faut plusieurs setw(x)).

```
char nom[20], prenom[20];
cin » setw(20) » nom » setw(20) » prenom;
```

Ici je ne pourrai saisir que des chaînes de longueur max 19.

Formatages à l'aide de manipulateurs (2/2)

Formatages du flot d'entrée *cin* :

- setw(x) : lire x caractères au maximum pour la prochaine chaîne de style C (si plusieurs chaînes de style C, alors il faut plusieurs setw(x)).

```
char nom[20], prenom[20];
cin » setw(20) » nom » setw(20) » prenom;
```

Ici je ne pourrai saisir que des chaînes de longueur max 19.

Plan

Les opérateurs new et delete

En C++, on utilise **new** à la place du malloc de C et **delete** à la place du free de C.

```
#include <cstdlib> // Pour la définition du symbole EXIT_SUCCESS

int main() // le programme principal, le point d'entrée du programme
{
   double* x = new double; // allocation d'un élément de type double
   delete x; // libération mémoire d'un élément

   float* f = new float(1.5f); // allocation d'un élément de type float initialisé à 1.5
   delete f; // libération mémoire d'un élément

   long* tab = new long[100] ; // allocation d'un tableau de 100 long ; noter les [ ]

   delete [] tab; // libération mémoire d'un tableau : ATTENTION au []

   return EXIT_SUCCESS;
}
```

Les opérateurs new et delete

- **new** *type* ou **new** *type*[n] fournissent comme résultat un pointeur de type *type** si l'allocation a réussi.
- En cas d'échec, l'opérateur **new** déclenche une **exception** de type *bad_alloc*. Si l'exception n'est pas traitée par le programmeur, le programme s'interrompt (cf. cours sur les exceptions). Pour éviter ce pb : **new** (std: :nothrow) *type*.
- Le comportement du programme n'est pas défini si on libère par l'opérateur **delete** :
  - un emplacement mémoire déjà libéré.
  - une mauvaise adresse (obtenue autrement que par l'opérateur **new**, e.g. par malloc).
- La syntaxe **delete** [] *adresse* n'est obligatoire que pour les tableaux d'objets (cf. cours sur les objets).

Les opérateurs new et delete

- **new** *type* ou **new** *type*[n] fournissent comme résultat un pointeur de type *type** si l'allocation a réussi.
- En cas d'échec, l'opérateur **new** déclenche une **exception** de type *bad_alloc*. Si l'exception n'est pas traitée par le programmeur, le programme s'interrompt (cf. cours sur les exceptions). Pour éviter ce pb : **new** (std: :nothrow) *type*.
- Le comportement du programme n'est pas défini si on libère par l'opérateur **delete** :
  - un emplacement mémoire déjà libéré.
  - une mauvaise adresse (obtenue autrement que par l'opérateur **new**, e.g. par malloc).
- La syntaxe **delete** [] *adresse* n'est obligatoire que pour les tableaux d'objets (cf. cours sur les objets).

Les opérateurs new et delete

- **new** *type* ou **new** *type*[n] fournissent comme résultat un pointeur de type *type** si l'allocation a réussi.
- En cas d'échec, l'opérateur **new** déclenche une **exception** de type *bad_alloc*. Si l'exception n'est pas traitée par le programmeur, le programme s'interrompt (cf. cours sur les exceptions). Pour éviter ce pb : **new** (std: :nothrow) *type*.
- Le comportement du programme n'est pas défini si on libère par l'opérateur **delete** :
  - un emplacement mémoire déjà libéré.
  - une mauvaise adresse (obtenue autrement que par l'opérateur **new**, e.g. par malloc).
- La syntaxe **delete** [] *adresse* n'est obligatoire que pour les tableaux d'objets (cf. cours sur les objets).

Les opérateurs new et delete

- **new** *type* ou **new** *type*[n] fournissent comme résultat un pointeur de type *type** si l'allocation a réussi.
- En cas d'échec, l'opérateur **new** déclenche une **exception** de type *bad_alloc*. Si l'exception n'est pas traitée par le programmeur, le programme s'interrompt (cf. cours sur les exceptions). Pour éviter ce pb : **new** (std: :nothrow) *type*.
- Le comportement du programme n'est pas défini si on libère par l'opérateur **delete** :
  - un emplacement mémoire déjà libéré.
  - une mauvaise adresse (obtenue autrement que par l'opérateur **new**, e.g. par malloc).
- La syntaxe **delete** [] *adresse* n'est obligatoire que pour les tableaux d'objets (cf. cours sur les objets).