



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 1. Du C au C++

Version du 3 septembre 2024



Ce travail est publié sous licence Creative Commons
Paternité – Pas d'utilisation commerciale – Pas de modification

Chapitre 1

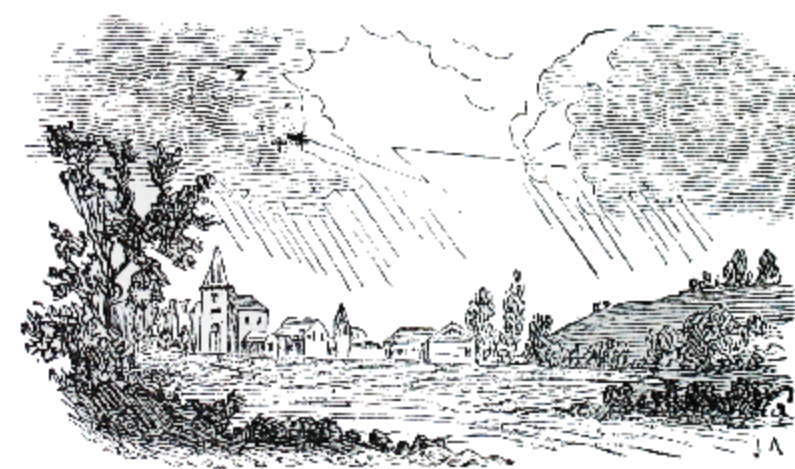


Fig. 75. — Le plus souvent la décharge a lieu entre deux nuages.

Du C au C++

Le langage C++ est littéralement un « C augmenté ». Mais les ajouts ne concernent pas uniquement le concept d'objet qui sera traité au chapitre 2. Dans le présent chapitre, on s'intéresse aux particularités du langage qui le distinguent du C mais qui ne sont pas pour autant liées à la *Programmation Orientée Objet* (POO).

1.1 Parmi les nouveautés

1.1.1 Fichiers d'en-têtes

La bibliothèque standard C++ englobe celle du C. Les fichiers d'en-tête du C sont donc utilisables (`<stdio.h>`, `<string.h>`, etc.). Des en-têtes supplémentaires fournissent les déclarations et définitions propres à la bibliothèque C++. Avant la norme de 1998 ils portaient des noms comme `<iostream.h>` ou `<fstream.h>`. Toutefois, suite à l'introduction des espaces de noms (§ 1.2.2) avec la norme [3], des équivalents aux fichiers d'en-têtes de la bibliothèque standard ont été ajoutés pour assurer la compatibilité ascendante. Ainsi :

```
<iostream.h> devient <iostream>,  
<fstream.h>  devient <fstream>, etc.
```

L'inclusion de la version `.h` assure un comportement identique à une en-tête sans espace de noms, alors que la version sans `.h` est compatible avec la norme et place les identificateurs dans l'espace de noms standard `std`.

De même, tous les fichiers d'en-têtes de la bibliothèque standard C ont un équivalent, sans `.h` et préfixé par la lettre 'c', qui place lui aussi les identificateurs dans l'espace de noms `std`.

```
<limits.h>  devient <climits>,  
<stdlib.h>  devient <cstdlib>,  
<string.h>  devient <cstring>,  
<stdio.h>   devient <cstdio>, etc.
```

Par exemple, l'en-tête `<cstdio>` déclare la fonction `std::printf` au lieu de `printf` comme le fait `<stdio.h>`.

Remarque 1.1 *Le fichier d'en-tête `<limits>` de la librairie C++ remplit entre autres le rôle*

joué par le fichier `<limits.h>` de la bibliothèque C mais avec une syntaxe propre au C++. Ceci est détaillé au chapitre 8. Il est important de noter que son nom n'est pas préfixé par un 'c', et que l'en-tête `<climits>` existe aussi.

1.1.2 Commentaires

Comme en C99¹, la séquence de caractères « `//` » permet de mettre tout ce qui la suit, et jusqu'à la fin de ligne, en commentaire (cf. listing 1.1).

```
1 {
2     float x = 1.414f; // Approximation of the square root of 2.
3     x = sqrt( 2.0f /* 2 as a float */ );
4 }
```

Listing 1.1 – Commentaires C++.

1.1.3 Structures

La déclaration de variables de types composés (structures) ne nécessite plus la répétition du mot-clé **struct**. Le nom de la structure suffit.

```
1 struct SComplex {
2     double re, im;
3 };
4 struct SComplex z;
5 typedef struct SComplex Complex;
6 Complex r = { 0.0, 1.0 };
```

Listing 1.2 – Structures en C.

```
1 struct Complex {
2     double re, im;
3 };
4 Complex r;
```

Listing 1.3 – Structures en C++.

1.1.4 Types de base

Mis à part les types *référence* (§ 1.5) et booléen (**bool**), ce sont les mêmes que ceux du langage C². De plus, la norme 2011 du langage (C++11) introduit le type **long long** (**int**) que plusieurs compilateurs proposaient déjà. Le tableau 1.1 donne les désignations et domaines de ces types de base.

Les tailles des différents types ne sont pas précisées par la norme et sont dépendantes des implémentations (i.e., architecture ou compilateur). Par contre, la norme précise certaines relations d'ordre entre ces tailles. Dans ce qui suit, \preceq signifie « occupe au plus autant d'espace que » et \doteq signifie « possède la même taille que ».

`sizeof(char)==1` (mais ≥ 8 bits!)
`char` \preceq `short` \preceq `int` \preceq `long` \preceq `long long`
`1` \preceq `bool` \preceq `long` `2` octets \preceq `short` `4` octets \preceq `long` `8` octets \preceq `long long`
`char` \preceq `wchar_t` \preceq `long` `float` \preceq `double` \preceq `long double`
 Pour tout type entier T , on a $T \doteq \text{signed } T \doteq \text{unsigned } T$.

1. Pour être précis, en C90 amendement AMD1.

2. La norme C99 introduit les booléens **true** et **false** ainsi que le type **bool** sous forme de macros via l'entête `<stdbool.h>`. En C++, ce sont des mots-clés.

Désignation	Mot-clé	Exemples littéraux
booléen	<code>bool</code>	<code>true</code> , <code>false</code>
caractère	<code>char</code>	<code>'a'</code> , <code>'\n'</code> , <code>'0'</code> , 48, 117, 234 <i>ou</i> -122
caractère large	<code>wchar_t</code>	<code>'ab'</code> , 2345
entier	<code>short (int)</code> , <code>int</code> , <code>long (int)</code> et <code>long long (int)</code>	1, -256, -1234
virgule flottante	<code>float</code> , <code>double</code> et <code>long double</code>	-1.06e-8, 1.4, 255.0f (double par défaut)
énuméré	<code>enum</code>	
indéfini / inexistant	<code>void</code>	<code>void foo();</code> <code>void *p;</code>
pointeur	$T *$	
tableau	$T []$	<code>{true, false, true}</code>
référence	$T \&$	

TABLE 1.1 – Types de base du langage.



La taille des types *intégral* (c.-à-d. entiers, dont `char`, et booléens) dépend de l'implémentation. Généralement, c'est 4 pour un `int` sur une architecture 32 ou 64 bits, mais ce n'est pas du tout une obligation.

Pour connaître les intervalles de valeurs possibles des types numériques, il est possible d'utiliser les constantes définies dans les fichiers d'en-têtes de la bibliothèque C. En effet :

`limits.h` définit `INT_MAX`, `UINT_MAX`, etc.
`float.h` définit `FLT_MAX`, `DBL_MAX`, `DBL_EPSILON`, etc.

En C++, on pourra utiliser un code ressemblant à celui du listing 1.4 sur lequel nous reviendrons dans la section 8.1.

```
1 int iMax = std::numeric_limits<int>::max();
```

Listing 1.4 – La plus grande valeur possible pour un entier (`int`) en C++.

La syntaxe précédente est sans doute obscure pour qui ne connaît pas encore :

- les espaces de noms ;
- les notions de classe, de modèle de classe et de spécialisations ;
- la notion de méthode statique.

Le cas du pointeur nul

En langage C, la constante pré-processeur `NULL` est parfois définie de la manière suivante :

```
1 #define NULL ((void*)0)
```

C'est tout à fait convenable en C puisque les pointeurs sur `void` sont convertis implicitement, dans ce langage, en tout type de pointeur. Mais ce n'est pas le cas en C++. Ainsi, le code C++ du listing 1.5 provoque une erreur de compilation car le pointeur `NULL` (de type `void*`) ne peut pas être converti implicitement en un `const char*`.


```

1 // Possible definition in C
2 // #define NULL ((void*)0)
3
4 void foo(const char *) {
5     // ...
6 }
7
8 int main() {
9     foo(NULL);
10 }

```

Listing 1.5 – Code erroné du fait de la définition de NULL.

```

1 // Possible definition of NULL
2 #define NULL 0
3
4 void foo(int i) {
5 }
6 void foo(const char *) {
7 }
8
9 int main() {
10     foo(0);           // Call foo(int)
11     foo(NULL);        // Same thing!
12     foo((const char *)0); // Finally!
13 }

```

Listing 1.6 – Utilisation malheureuse du pointeur NULL.

Le C++ apporte une solution : la valeur constante 0 est une valeur constante valide pour tout type de pointeur. Il faut donc préférer son utilisation à celle de la macro NULL.

C++11 : le mot-clé `nullptr`

Le double sens de la constante littérale 0 (à la fois entier et pointeur nul) pose un problème lors de l'appel d'une fonction surchargée. En effet, pour peu que la macro NULL soit définie comme suit :

```

1 #define NULL (0)

```

le code du listing 1.6 n'aboutira pas au résultat escompté (lignes 10 et 11).

Pour remédier à cet inconvénient, le C++11 introduit le mot-clé `nullptr` qui est une constante de type `nullptr_t`, pour laquelle une conversion implicite en n'importe quel type de pointeur est possible, ainsi qu'en la valeur booléenne `false`. Comme ce sont là les seules conversions possibles de cette constante, elle permet de lever toute ambiguïté. Ainsi, dans le listing 1.6, l'utilisation de `nullptr` (ligne 12) en lieu et place de NULL provoquera l'appel de la seconde fonction `foo()`.

```
1 | int main() {  
2 |     int i, j;  
3 |     cin >> i >> j;  
4 |     unsigned long n;  
5 |     cin >> n;  
6 |     i = (int) n;  
7 | }
```

Listing 1.7 – Déclaration de variables au sein d’un bloc.

1.1.5 Position des déclarations de variables

En C++ comme en C99, les déclarations de variables ne se font pas nécessairement en début de bloc avant toute instruction, mais elle peuvent se faire *entre* deux instructions (cf. listing 1.7). Dans tout les cas, la portée d’une variable s’étend de la fin de sa déclaration jusqu’à la fin du bloc.

1.1.6 Fonctions sans arguments

Il existe une différence, qui peut paraître anodine, entre les fonctions sans arguments déclarées en C par rapport au C++. En effet, le code C ci dessous :

```
| float sum();
```

signifie « il existe une fonction de nom **sum** retournant un **float** et qui accepte des arguments, sans que le nombre et le type des arguments ne soient précisés dans cette déclaration. Le prototype d’une fonction sans arguments s’écrit, toujours en C :

```
| float random(void); // This is C (and not C++)
```

En C++, par contre, le prototype ci-dessous est, sans aucune ambiguïté, celui d’une fonction qui n’accepte *aucun* argument.

```
| float random(); // Two distinct meanings between C and C++
```

Finalement, on pourra bannir en C++ l’utilisation de **void** comme synonyme de l’absence d’arguments [10].

1.1.7 Valeurs d’arguments par défaut

Les paramètres de fonction peuvent avoir des valeurs par défaut. Le type est vérifié lors de la déclaration, qui doit être *unique*, et la valeur par défaut est évaluée au moment de l’appel de la fonction. La valeur par défaut peut être une variable globale ou une expression faisant intervenir des variables globales. Attention : la valeur par défaut doit être précisée dans la déclaration de la fonction mais pas dans sa définition (cf. listing 1.8).

Règle Si un argument possède une valeur par défaut, tous ceux qui le suivent dans la déclaration aussi. [...]

```

1 // Compute a price ‘‘All Taxes Included’’
2 // (declaration in "taxes.h")
3 float priceATI(float price, float vat = 18.6f);
4
5 // (definition in "taxes.cpp")
6 float priceATI(float price, float vat = 18.6f) { // Error!
7     return price * (1 + vat / 100.0f);
8 }

```

Listing 1.8 – Arguments de fonction par défaut.

1.1.8 Boucle for et portée des déclarations

```

1 int a, b;
2 for (int i = 0; i < 10; ++i) {
3     // Body of the loop
4     // ...
5 }

```

Listing 1.9 – Déclaration de variable dans une instruction for.

Dans le cas du listing 1.9, la variable `i` n’est visible que dans le corps de la boucle. Dans ce corps, elle masque bien entendu toute variable de même nom définie dans le bloc de niveau supérieur.

1.1.9 Définitions : objets et lvalues

Dans la suite, on conviendra d’appeler :

- **objet**, une zone contiguë de mémoire (\neq instance de classe) ;
- **lvalue**, une expression faisant référence à un objet en mémoire (et dont on peut récupérer l’adresse avec l’opérateur `&`) ;
- **lvalue modifiable**, une expression faisant référence à un objet non constant ;
- **rvalue**, une expression qui n’est pas une *lvalue*, ou un objet temporaire qui n’existe que durant l’évaluation d’une expression.

1.2 Portée et espaces de noms

1.2.1 Opérateur de résolution de portée

Rappel En C++ les déclarations « volantes » sont possibles.

Lorsqu’une variable définie localement à un bloc masque une variable définie en dehors de ce bloc (p. ex. une variable globale), l’opérateur de *résolution de portée* `::` utilisé sans préfixe permet de faire référence à la variable masquée (cf. listing 1.10).


```

1  int i;
2  void someFunction() {
3      int x = i; // global variable 'i'
4      cout << "x=" << x << endl;
5      int i = 0; // Declaration of a local 'i' (masking)
6      i = 10 * i; // Set and use the local 'i'
7      ::i = 100; // Set the global 'i'
8  }

```

Listing 1.10 – Résolution de portée.

1.2.2 Espaces de noms

Toute portion de code de niveau global peut être placée dans un espace de noms. Il peut s'agir d'une simple déclaration comme d'un fichier complet. La syntaxe est donnée par le listing 1.11. Dans cet exemple, l'identificateur `inputMatrix` situé entre les accolades est alors défini dans l'espace de noms `my_lib_name`. Cela revient à dire que d'un point de vue global son nom est en fait `my_lib_name::inputMatrix`.

```

1  namespace my_lib_name {
2      // Declarations of variables
3      // Definitions of types (e.g. typedef)
4      // Definitions of classes, functions, etc.
5      // For example :
6      void inputMatrix() { /* ... */ };
7  }

```

Listing 1.11 – Code placé dans un espace de noms.

Afin d'alléger la syntaxe, il est possible d'importer un symbole depuis un espace de noms vers la portée courante à l'aide du mot-clé `using`. L'importation est alors valable pour le reste de l'unité de compilation ou du bloc courant, comme dans l'exemple du listing 1.12. On parle dans ce cas de *using-declaration*.

```

1  #include <iostream>
2
3  namespace mathematics
4  {
5      const double e = 2.71828182845904523536;
6  }
7
8  using mathematics::e;
9
10 std::cout << e << std::endl;

```

Listing 1.12 – Import d'un symbole dans l'espace de noms global.

Il est aussi possible d'importer *tout* le contenu d'un espace de noms dans l'espace courant à l'aide de la directive `using namespace` selon le modèle du listing 1.13 pour l'utilisation des identifiants `cout` et `endl` de l'espace de noms `std`. On parle alors de *using-directive*.

```

1 #include <iostream>
2 namespace mathematics
3 {
4     const double e = 2.718;
5     const double pi = 3.1416;
6 }
7
8 using namespace std;
9
10 int main() {
11     cout << mathematics::e << endl;
12 }

```

Listing 1.13 – Import « massif » de symboles dans l'espace de noms global.

```

1 #include <iostream>
2 namespace Earth {
3     float acceleration = 9.81;
4 }
5 namespace Moon {
6     float acceleration = 1.622;
7 }
8 int main(int , char *[])
9 {
10     using namespace Earth;
11     using namespace Moon;
12     std::cout << acceleration << std::endl; // Ambiguous!
13     return 0;
14 }

```

Listing 1.14 – Symbole ambigu suite à deux *using-directives*.

L'import est valable pour le reste du bloc dans lequel la directive (c.-à-d. `using namespace`) ou la déclaration (`using`) est placée. Si elle est placée en dehors de tout bloc, l'import est valable pour la suite de l'unité de compilation.

Conflits Dans certaines situations, des conflits de noms peuvent résulter d'imports réalisés par des *using-declarations* ou des *using-directives*. S'ensuivent alors des ambiguïtés au moment de l'utilisation des symboles (cf. listing 1.14). Toutefois, un symbole déclaré classiquement dans un bloc ou bien via une *using-declaration* est prioritaire sur le même symbole importé via une *using-directive* (cf. listing 1.15).

On tiendra compte de la mise en garde suivante :



La directive `using namespace` ne devrait jamais être utilisée dans les fichiers d'en-têtes en dehors de tout bloc. Pourquoi ?

```

1  #include <iostream>
2  namespace Earth {
3      float acceleration = 9.81;
4  }
5  namespace Moon {
6      float acceleration = 1.622;
7  }
8  namespace Sun {
9      int position = 0;
10 }
11 int main(int , char *[])
12 {
13     using Moon::acceleration;
14     using namespace Earth;
15     std::cout << acceleration << std::endl; // Moon: 1.622
16
17     int position = 20;
18     using namespace Sun;
19     std::cout << position << std::endl; // 20, not 0
20     return 0;
21 }

```

Listing 1.15 – Priorité d'un (*using-*)*declaration* sur une *using-directives*.

Un dernier exemple est donné dans le listing 1.16.

Quelques remarques finales :

- Les espaces de noms peuvent être imbriqués, avec modération. [...]
- Un espace de noms sans nom (ou anonyme) est toujours local à une unité de compilation. Il s'agit là de la version C++ des déclarations `static` de variables globales et de fonctions (cf. section 1.3.4).
- Les en-têtes `<c***>` fournies avec le compilateur GNU `g++` ne placent pas les symboles de la bibliothèque C de manière exclusive dans l'espace de noms `std`. Elles les placent aussi dans l'espace de nom global. C'est une liberté laissée par la norme.
- Concrètement, les bibliothèques du compilateur précédemment cité réalisent l'import inverse : espace de noms global vers l'espace `std`, à l'aide de *using-declarations*. La fonction `printf` est ainsi disponible à la fois dans l'espace global *et* dans l'espace `std`.

1.3 Portée et compilation séparée (C et C++)

1.3.1 Compatibilité des fichiers d'en-têtes

Les fichiers d'en-têtes standards du C++ (et pas du C, comme par exemple `iostream.h`) qui existaient dans les versions précédant la norme de 1998 sont conservés et permettent de régler le problème de la compatibilité ascendante. En effet, la différence entre un fichier inclus comme `<iostream>` et `<iostream.h>` est que ce dernier importe dans l'espace de noms global tous les symboles qui, dans `<iostream>`, se trouvent uniquement dans l'espace de noms `std`.

```

1 namespace Moon
2 {
3     const double acceleration = 1.622;
4     double mass2weight(double mass) {
5         return mass * acceleration;
6     }
7     // ...
8 } // namespace Moon
9 namespace Earth
10 {
11     const double acceleration = 9.81;
12     double mass2weight(double mass) {
13         return mass * acceleration;
14     }
15 } // namespace Earth
16 double my_weight = Earth::mass2weight(80.0);
17 double acc = Moon::acceleration;
18 using namespace Moon; // Or [...]
19 accel = acceleration;

```

Listing 1.16 – Exemple d'utilisation des espaces de noms.

Ainsi, la directive d'inclusion suivante :

```
#include <iostream.h>
```

est équivalente à

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

Le fichier `<iostream>` étant semblable au listing 1.17.

```

1 #ifndef _GLIBCXX_IOSTREAM
2 #define _GLIBCXX_IOSTREAM 1
3 // ... Several include directives
4 namespace std
5 {
6     // Declarations [...]
7 }
8 #endif /* _GLIBCXX_IOSTREAM */

```

Listing 1.17 – Fichier `<iostream>`

1.3.2 Portées

Remarque 1.2 Une variable définie dans un fichier (c.-à-d. une unité de compilation) en dehors de toute fonction, bloc ou classe a une portée de fichier.

Remarque 1.3 *Un objet doit être défini au plus une fois dans un même programme. Ainsi, le code des listings 1.20 et 1.19 est incorrect. En effet, la variable `g` est définie deux fois et provoquera une erreur lors de l'édition de liens.*

```
1 double mass2weight(double);
```

Listing 1.18 – Gravity.h

```
1 #include "Gravity.h"
```

```
2
3 // ...
```

```
4
5 double g = 9.81;
```

```
6
7 double mass2weight(double) {
8     // ...
9 }
```

Listing 1.19 – Gravity.cpp

```
1 #include <iostream>
```

```
2 #include "Gravity.h"
```

```
3 using namespace std;
```

```
4
5 double g = 9.81;
```

```
6
7 int main() {
8     cout << mass2weight(80);
9     cout << endl;
10    return 0;
11 }
```

Listing 1.20 – Rocket.cpp

1.3.3 Mot-clé extern

Le mot-clé `extern` permet de déclarer, *sans le définir*, un objet défini dans une autre unité de compilation. Il permet, intuitivement, d'exprimer l'*existence* dans le programme complet d'un objet d'un type donné. L'exemple de la remarque 1.3 peut être compilé sans erreur si on remplace dans `Rocket.cpp` :

```
5 double g = 9.81; // Declaration and definition.
```

par

```
5 extern double g; // Declaration only.
```

Mais cette correction reste généralement insatisfaisante. Il est en effet logique dans ce cas que la déclaration `extern` soit faite dans le fichier `Gravity.h`, et uniquement dans celui-là.

1.3.4 Mot-clé static et espace de noms anonyme

Pour rappel, en langage C, le mot-clé `static` appliqué à un objet ayant une portée de fichier rend cet objet « réellement » local. Mais...

« Évitez ce mot-clé en dehors des fonctions et des classes. »

Bjarne Stroustrup, [9]

En effet, en C++ une construction beaucoup plus flexible existe : les espaces de noms anonymes. On pourra par exemple utiliser le code du listing 1.21 pour rendre la variable `g` réellement locale au fichier `Rocket.cpp` (comme l'aurait fait le mot-clé `static`). Dans ce cas, `g` ne peut plus entrer en conflit avec un symbole de même nom défini dans une autre unité de compilation.


```
1 #include <iostream>
2 #include "Gravity.h"
3 using namespace std;
4 namespace {
5     double g = 9.81;
6 }
7 int main() {
8     cout << mass2weight(80);
9     cout << endl;
10    return 0;
11 }
```

Listing 1.21 – Espace de noms anonyme.

Cette syntaxe est préférable car plus polyvalente : contrairement au mot-clé `static`, un espace de nom anonyme peut aussi s'appliquer à un type défini par l'utilisateur (classe, structure, `typedef`) pour rendre ce type local lui aussi.

1.4 Surcharge de fonctions

Principe Un même nom peut être utilisé pour définir plusieurs fonctions. Dans ce cas, chaque fonction se distingue par le type de ses arguments. Attention, le type retourné n'est donc pas pris en compte.

1.4.1 Exemples de déclarations

```
1 double inverse(double x);
2 Complex inverse(Complex z);
3
4 int abs(int i);
5 double abs(double x);
6 double abs(Complex z);
7
8 char succ(char c);
9 int succ(int i);
```

Listing 1.22 – Surcharge de fonctions : déclaration.

1.4.2 Appel de fonction surchargée

Au moment de l'appel d'une fonction surchargée, le compilateur doit choisir la bonne fonction. Il le fait en examinant les possibilités suivantes, classées par ordre de priorité :

- correspondance exacte des types ;
- application possible de la promotion (p. ex. : `short` \rightarrow `int`) ;
- application possible des conversions standards (p. ex. : `int` \leftrightarrow `double`) ;
- application de conversions définies par le programmeur ;

— utilisation des points de suspension (...).

La première règle qui s'applique est utilisée, mais il peut arriver qu'une des cinq règles laisse plusieurs choix! [...] Dans ce cas, la compilation échoue et les règles qui suivent ne sont pas examinées.

Ainsi, en supposant que les déclarations du listing 1.22 sont présentes, on peut examiner le code suivant :

```

1 float s = 10.0f;
2 double w = 2.0;
3 float z;
4
5 z = inverse(w); // Call inverse(double)
6 z = inverse(s); // Cast s as double,
7                  // then call inverse(double)
8 int i = succ(z); // Cast z as int, then call succ(int)

```

Listing 1.23 – Surcharge de fonction : l'appel.



La résolution de surcharge ne se fait qu'au sein d'une même portée (corps de fonction, classe, bloc) à l'exception des espaces de noms pour lesquels il est possible d'intervenir à l'aide de directives ou déclarations *using*.



1.5 Références

Les références constituent une notion nouvelle et très utile du langage. En termes simples, une référence est un synonyme pour un objet, on parle aussi parfois d'*alias*. Un des intérêts majeurs de cette notion est l'allègement de la syntaxe du passage d'arguments par adresses (§ 1.5.3).

1.5.1 Déclaration et initialisation dans un bloc

Étant donné un identificateur de type T , la syntaxe de déclaration d'une référence est la suivante :

$$\begin{array}{l}
 T \ y; \\
 T \ \& \ x = y;
 \end{array}$$

Dans ce cas l'opérateur n'initialise pas une *variable* x avec la valeur de y mais initialise x comme une *référence* à y . Les points suivants sont importants et à retenir :

- Une référence déclarée dans un bloc *doit* être initialisée lors de sa déclaration.
- Une référence déclarée et initialisée de cette façon est *définitive*.
- L'adresse d'une référence est celle de la variable référencée.

```

1 #include <iostream.h>
2
3 void linear_transform(float & x, float s) {
4     x *= s;
5 }
6
7 int main() {
8     float y = 2.0, z = 10;
9     cin >> y;
10    linear_transform(y, z);
11    cout << "y = " << y << endl;
12    return 0;
13 }

```

Listing 1.25 – Passage par adresse en C++.

Exemple

<pre> accel: g: alpha: pg: </pre>	<table border="1" style="border-collapse: collapse; width: 100px; text-align: center;"> <tr><td style="height: 20px;"></td></tr> <tr><td style="height: 20px;">9.81</td></tr> <tr><td style="height: 20px;">0.6e-9</td></tr> <tr><td style="height: 20px;">&g</td></tr> <tr><td style="height: 20px;"></td></tr> </table>		9.81	0.6e-9	&g		<pre> double double double* </pre>
9.81							
0.6e-9							
&g							

```

1 {
2     double g = 9.81;
3     double & accel = g;
4     double alpha = 0.6e-9;
5     double * pg;
6     accel = 1.0; // g is 1.0
7     pg = &accel; // value of pg ?
8 }

```

Listing 1.24 – Déclaration et initialisation de références dans un bloc.

1.5.2 Références comme arguments de fonctions

Utilisées comme paramètres de fonctions, les références offrent les avantages du « passage par adresse » possible en C à l'aide de pointeurs, mais avec une notation allégée. Dans la fonction `trans_lineaire()` du listing 1.25, aucun espace n'est réservé sur la pile pour `y` stocker la variable `x` (comme c'est le cas, rappelons le, pour `s` qui est une *copie* de la variable `z` donnée en argument). La référence `x` s'utilise comme une *lvalue* normale, et toute opération sur `x` porte en fait sur le `y` de la fonction `main`.

Exercice 1.1 Écrire l'équivalent en C du listing 1.25.



A l'appel, le passage par référence est moins explicite que le passage par adresse réalisé grâce aux pointeurs (cf. ligne 10 du listing 1.25). Attention donc aux noms des fonctions, et pensez aux commentaires !


1.5.3 Référence comme type de retour d'une fonction

Il est possible pour une fonction de retourner une référence. C'est intéressant car un appel de fonction peut ainsi être une *lvalue*. Cela permet aussi à une fonction de transmettre à la fonction appelante une référence qui a été reçue comme argument (cf. la surcharge de l'opérateur << par une fonction à deux arguments). Seule la syntaxe de la déclaration de fonction est différente par rapport à un type de retour classique (cf. listing 1.26). L'instruction **return** ne change pas et sert à désigner la *lvalue* sur laquelle porte la référence qui doit être retournée.

```

1 | int & max(int & x, int & y) {
2 |     return (x >= y) ? x : y;
3 | }
4 |
5 | int main() {
6 |     int a = 10, b = 20;
7 |     max(a, b) += 10; // ?
8 | }
```

Listing 1.26 – Fonction dont le type de retour est une référence.

Dans le cas du listing 1.26, les variables **x** et **y** sont elles mêmes des références, mais il pourrait s'agir par exemple de cellules particulières d'un paramètre de type tableau d'entiers. 

Première illustration Saisie clavier et opérateur ».

L'opérateur >> surchargé pourrait avoir la déclaration suivante³ :

```
| ifstream & operator>>( ifstream & in, int & i );
```

Par conséquent, l'instruction de la ligne 2 suivante

```

1 | Quaternion a, b;
2 | cin >> a >> b;
```

est équivalente à :

```
| operator>>( operator>>( cin, a ), b );
```

Remarque 1.4 L'opérateur >> est associatif de gauche à droite.

Seconde illustration Premier élément non nul d'un tableau.

```

1 | int & first_non_zero(int array[], int n) {
2 |     for (int i = 0; i < n; i++) {
3 |         if (array[i]) {
4 |             return array[i];
5 |         }
6 |     }
7 |     throw "No non-zero value in array";
8 | }
```

Listing 1.27 – Référence sur le premier élément non nul d'un tableau.

3. La norme le définit en fait comme une méthode (cf. section 4.1).

Les instructions `return` de cet exemple renvoient une référence, pas une valeur. On peut donc écrire :

```
|| first_non_zero( an_array, 10 ) = 0;
```



On ne retourne jamais une référence à un objet local à une fonction, sauf si c'est une variable « `static` ». (Pourquoi ?)

1.5.4 Références comme données membres d'une classe

Ce sujet sera logiquement abordé une fois que les classes en C++ l'auront été.

1.5.5 Références constantes et objets temporaires

Il n'est pas possible d'initialiser une référence avec une constante littérale. Le code suivant n'est donc pas autorisé :

```
|| int & i = 45; // Forbidden, 45 is not an lvalue
```

Toutefois, l'initialisation d'une référence déclarée constante est dans ce cas possible :

```
|| const int & i = 45; // Correct
```

En effet, la ligne précédente peut être interprétée comme suit :

```
1 | int temporaryAndAnonymous = 45;
2 | const int & i = temporaryAndAnonymous;
```

Il y a donc création d'un objet temporaire et anonyme. Qui plus est, l'objet référencé peut alors être de type différent, et les règles de conversion s'appliquent. Ainsi,

```
1 | int i = 10;
2 | const double & phi = i;
```

est équivalent à

```
1 | int tmp1 = 10;
2 | double tmp2 = tmp1; // Implicit cast
3 | const double & phi = tmp2;
```

1.5.6 Objets temporaires

Soit la déclaration :

```
|| Complex z = x + v * r;
```

Un objet (structure ou instance de classe) temporaire est créé pour stocker le résultat de l'expression `v * r`. Un tel objet n'est pas une *lvalue* et ne peut donc pas être utilisé pour initialiser une référence, sauf si c'est une référence constante (cf. listing 1.28).


```

1 void f(double & x) { /* [...] */ }
2 void g(const double & x) { /* [...] */ }
3
4 double x = 1.0, y = 2.0;
5 f(x + y);    // Error
6 f(3.14159);  // Error
7 g(9.81);     // OK

```

Listing 1.28 – Références à des objets temporaires.

Portée des objets temporaires

« Une variable temporaire utilisée dans l’initialisation d’une référence [constante] persiste jusqu’à la fin de la portée de sa référence. »

Bjarne Stroustrup, [9]

1.6 Allocation dynamique sur le tas

Les fonctions `malloc`, `calloc`, `realloc` et `free` de la bibliothèque C⁴ peuvent toujours être utilisées en C++; mais on doit leur préférer l’utilisation des deux nouveaux mots-clés `new` et `delete` dont la syntaxe est présentée dans cette section. En effet, comme il sera expliqué dans la section 2.1.4, le mot clé `delete` provoque, dans le cas d’une instance de classe, l’appel d’une méthode particulière appelée *destructeur*. La fonction `free` de la bibliothèque C n’offre pas cette possibilité.

1.6.1 Allocation

Pour toute dénomination de type T , l’expression :

`new T;`

alloue `sizeof(T)` octets et a pour valeur, en cas de succès, l’adresse du premier octet sous la forme d’un pointeur *typé* (de type T^*). Elle est implémentée à l’aide de l’opérateur pouvant être surchargé « `void *operator new(size_t);` ».

D’autre part, l’expression

`new T[n];`

alloue la mémoire nécessaire pour n objets de type T et vaut l’adresse du premier objet (si tout s’est bien passé). Bien entendu, n peut être une constante mais aussi une expression à valeur entière. Attention : il n’y a pas d’initialisation automatique de la mémoire allouée (sauf pour un type objet auquel cas le constructeur par défaut est appelé pour chacune des instances créées). Mais il est possible de demander l’initialisation à zéro, pour les types de base, par la syntaxe suivante :

`new T[n]();`

4. En-tête `<cstdlib>`.

Exemples

```

1 char * pc = new char[255];
2 int * pi;
3 double * alpha;
4 double * e = new double;      // *e is not initialized
5 int * array = new int[10]();  // initialization with zeros
6 pi = new int[314];
7 alpha = new int[10];          // Error (alpha is not int*)

```

Listing 1.29 – Exemples d’allocations dynamiques.

En cas d’erreur

Si l’allocation est impossible, l’opérateur `new` soulève une exception standard dont le type est `std::bad_alloc` (cf. chapitre 6). Cependant, deux alternatives à la capture de l’exception `bad_alloc` sont offertes : l’installation d’un gestionnaire d’erreur ou l’utilisation de la valeur spéciale 0.

Installation d’une fonction gestionnaire d’erreur Si l’allocation est impossible (?), `new` peut faire appel à une fonction utilisateur, précisée à l’aide de la fonction `set_new_handler()` déclarée dans l’en-tête `<new>` (cf. listing ci-dessous).

```

1 #include <cstdlib>
2 #include <new>
3 void memory_error() {
4     std::exit(-1); // Rough
5 };
6
7 int main() {
8     std::set_new_handler(&memory_error);
9     int * pi = new int[0xFFFFFFFF];
10    std::exit(0);
11 }

```

Listing 1.30 – Gestion des erreurs d’allocation dynamique.

Utilisation de l’*allocator* `nothrow` L’*allocator* `nothrow` permet de revenir au comportement que l’opérateur `new` avait avant l’introduction des exceptions dans le langage, à savoir retourner 0 en cas d’échec. (A l’instar des fonctions `calloc()` et `malloc()` de la bibliothèque C.) La syntaxe est la suivante :

```

new(std::nothrow) T;
new(std::nothrow) T[n];

```

Remarque 1.5 Une particularité importante de l’opérateur `new` le distingue des deux fonctions `malloc` et `calloc` : son type de retour. En effet, la fonction `malloc` retourne un pointeur « `void*` » qui peut donc être converti implicitement, en C, en tout type de pointeur. En C++, l’expression `new T[10]` retourne un pointeur de type `T*`. Il est donc impossible, sauf en utilisant

```

1 struct Complex {
2     // ...
3 };
4
5 void foo() {
6     Complex * pz = new Complex;
7     int n;
8     std::cin >> n;
9     float * array = new float[n];
10
11     delete pz;
12     delete[] array;
13     delete pz; // ?
14 }

```

Listing 1.31 – Exemples d’allocations dynamiques et les désallocations respectives.

une conversion explicite⁵, d’allouer un tableau d’entier pour le stocker dans un pointeur de flottants!

1.6.2 Désallocation

L’instruction

```
delete pointeur;
```

libère la mémoire occupée par l’objet pointé, qui a été précédemment alloué par `new`. Elle est implémentée à l’aide de l’opérateur `void operator delete(void *)`;

Remarque 1.6 *On notera que :*

- Les instructions « `delete 0;` » et « `delete nullptr;` » sont valides (et sans effet).
- L’effet de « `delete p;` » est indéfini si `p` n’est pas valide (par exemple si `p` a déjà été désalloué).
- Le cas d’un pointeur sur une instance de classe est particulier (cf. section 2.1.4).

De plus, l’instruction

```
delete[] pointeur;
```

libère la mémoire occupée par un tableau d’objets précédemment alloué par `new[]`. Le cas d’un tableau d’instances de classes est aussi un cas particulier qui sera précisé dans la section 2.1.4.

1.6.3 Exemple

Des exemples d’allocations (dynamiques) et désallocations sont donnés dans le listing 1.31.

5. À l’aide de l’opérateur `reinterpret_cast<>()`.

Remarque 1.7 `new` et `delete` sont des opérateurs standards qui peuvent être surchargés par l'utilisateur (§ 2.2).



Ne pas mélanger `new/delete` avec `malloc()/free()`. Il faut choisir...