Alain Gibaud

Programmer en C++

Des premiers pas à la maîtrise de C++23

2e édition



ellipses

Programmer en C++

Références sciences

Programmer en C++

Des premiers pas à la maîtrise de C++23

2e édition

Alain Gibaud

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-105942
Dépôt légal : octobre 2025
©Ellipses Édition Marketing S.A.
8/10 rue la Quintinie 75015 Paris

DANGER LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle et artistique n'autorisant, aux termes des alinéas 2 et 3 de l'article L. 122-5, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1er de l'article L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

À Thérèse, à nos enfants et à leurs enfants.

J'exprime ma gratitude à ceux qui m'ont aidé sans le savoir, au moment où mon esprit vagabond a dû inventer les exemples qui illustrent ce livre : *Georges Brassens*, *Hergé*, *Patrick McGoohan*, *Arthur Rimbaud*, *Eric Rohmer*, *les producteurs de Pélardon des Cévennes*, *Simon and Garfunkel*, *Riri*, *Fifi* et *Loulou* qui se reconnaîtront. Sans oublier *Élio*, qui va bientôt apprendre le C++.

Steve Jobs et Dennis Richie moururent en 2011, à quelques jours d'intervalle.
Le moins que l'on puisse dire est que ces deux décès n'eurent pas le même écho dans l'espace médiatique.

Toutefois, un internaute commentant ces deux pertes fit sur un forum cette remarque lapidaire : « *En somme, c'est comme si Dennis Richie avait inventé l'ampoule électrique, et Steve Jobs la lampe de chevet* ».

Pour ceux qui ne le savent pas, Dennis Richie a inventé le langage C, qui a influencé l'immense majorité des langages actuellement utilisés. Il est aussi le co-auteur du système d'exploitation UNIX, dont sont dérivés de nombreux autres OS, dont MacOS.

Tout le monde sait que Steve Jobs était un entrepreneur éclairé.

Table des matières

I Préambule **25**

1 À propos de ce livre **27**
1.1 À qui s'adresse-t-il ? . . . 27
1.2 Que contient-il ? . . . 27
1.3 Les exemples . . . 27
1.4 Pictogrammes utilisés dans le texte . . . 30

2 À propos du langage C++ **31**

II Le socle de C++ **33**

3 Composition des programmes C++ **35**
3.1 La notion de fonction en C++ . . . 35
3.1.1 Invocation des fonctions . . . 35
3.2 La fonction `main` . . . 35

4 Les données et leur type **37**
4.1 Variables et constantes . . . 37
4.2 À quoi sert une déclaration ? . . . 37
4.3 Qu'est-ce qu'un type ? . . . 37
4.4 Les types primitifs . . . 38
4.4.1 Entiers . . . 38
4.4.2 Entiers techniques . . . 41
4.4.3 Booléens . . . 42
4.4.4 Réels . . . 42
4.4.5 Types des constantes . . . 44
4.4.6 Pointeurs . . . 47
4.5 Données constantes . . . 51
4.5.1 Qu'est-ce qu'une donnée constante ? . . . 51
4.5.2 Données constantes et constantes littérales . . . 51
4.5.3 Constantes énumérées . . . 52
4.5.4 `const` et les pointeurs . . . 52
4.5.5 Quand utiliser `const` ? . . . 53
4.6 Données volatiles . . . 54
4.6.1 Qu'est-ce qu'une donnée volatile ? . . . 54
4.6.2 Données volatiles et optimisations . . . 54

5 Opérateurs **55**
5.1 Opérateurs arithmétiques . . . 55
5.1.1 Opérateurs `+`, `-`, `*`, `/` . . . 55
5.1.2 Opérateur `%` . . . 55
5.1.3 Opérateurs `++` et `--` . . . 56
5.2 Opérateurs sur bits . . . 57
5.2.1 Opérateur `~` . . . 57
5.2.2 Opérateur `&` . . . 57
5.2.3 Opérateur `|` . . . 58
5.2.4 Opérateur `^` . . . 58

5.3 Opérateurs de décalage . . . 58
5.3.1 Opérateur `<<` . . . 58
5.3.2 Opérateur `>>` . . . 59
5.4 Opérateurs logiques . . . 59
5.4.1 Opérateur `!` . . . 60
5.4.2 Opérateur `&&` . . . 60
5.4.3 Opérateur `||` . . . 60
5.5 Opérateurs relationnels . . . 61
5.5.1 Opérateurs `<`, `>`, `<=`, `>=`, `==`, `!=` . . . 61
5.6 Opérateurs d'affectation . . . 61
5.6.1 Opérateur `=` . . . 61
5.6.2 Opérateurs de calcul « en place » . . . 62
5.7 Opérateurs relatifs aux pointeurs . . . 63
5.7.1 Opérateur de prise d'adresse (`&`) . . . 63
5.7.2 Opérateur de déréférencement (`*`) . . . 64
5.8 Opérateurs divers . . . 64
5.8.1 Opérateur conditionnel (`? :`) . . . 64
5.8.2 Opérateur de séquencement (`,`) . . . 64
5.8.3 Opérateur de transtypage (cast) . . . 65
5.9 Priorité et associativité des opérateurs . . . 66
5.9.1 Priorité des opérateurs . . . 66
5.9.2 Associativité des opérateurs . . . 67
5.10 Ordre d'évaluation des opérandes avant C++17 . . . 68
5.11 Ordre d'évaluation des opérandes à partir de C++17 . . . 68
5.12 Expressions hétérogènes et conversion automatique de type . . . 69
5.13 Promotion entière . . . 70

6 Fonctions **71**
6.1 Fonctions et sous-programmes . . . 71
6.1.1 Contexte informationnel des fonctions . . . 71
6.1.2 Définition d'une fonction . . . 71
6.1.3 Utilisation d'une fonction . . . 72
6.2 Paramètres réels et formels . . . 72
6.2.1 Mécanisme de passage des paramètres . . . 73
6.2.2 Paramètres d'entrée et de sortie . . . 73
6.2.3 Validité des paramètres réels . . . 74
6.3 Fonctions sans paramètre . . . 75
6.4 Sous-programmes (fonctions ne retournant rien) . . . 76
6.5 Fonctions à paramètres constants . . . 76
6.6 Fonctions `inline` . . . 77

7 Introduction aux entrées-sorties en C++ **79**
7.1 Qu'est-ce qu'une entrée ou une sortie ? . . . 79
7.2 Les entrées-sorties sur flux . . . 79
7.2.1 Le fichier d'en-tête `iostream` . . . 79
7.2.2 Affichage de données à l'écran . . . 79
7.2.3 Lecture de données au clavier . . . 81
7.2.4 Contrôle des entrées-sorties . . . 81
7.2.5 Entrées-sorties sur nouveaux types . . . 81

8 Organiser les données **83**
8.1 Tableaux . . . 83
8.1.1 Qu'est-ce qu'un tableau ? . . . 83
8.1.2 Déclaration d'un tableau . . . 83
8.1.3 Tableau de tableaux . . . 84
8.1.4 Initialisation des tableaux . . . 85
8.1.5 Manipulation des tableaux . . . 86
8.2 Chaînes de caractères héritées du C . . . 86
8.2.1 Initialisation des chaînes de caractères C . . . 87
8.2.2 Manipulation des chaînes de caractères C . . . 87
8.2.3 Entrées-sorties de chaînes de caractères C . . . 88
8.2.4 Jeux de caractères et encodage des littéraux . . . 89
8.3 Structures . . . 91
8.3.1 Qu'est-ce qu'une structure ? . . . 91
8.3.2 Déclaration d'un type structure . . . 91
8.3.3 Utilisation de structures . . . 91
8.3.4 Initialisation et affectation de structures . . . 92
8.3.5 Taille des structures . . . 93
8.4 Unions . . . 93
8.4.1 Qu'est-ce qu'une union ? . . . 93
8.5 Choisir entre un tableau et une structure . . . 94
8.6 Tableaux et structures comme paramètres de fonctions . . . 95

9 Instructions **97**
9.1 Les prédicats en C et C++ . . . 97
9.2 Instructions conditionnelles (tests) . . . 97
9.2.1 Forme générale de l'instruction `if .. else` . . . 97
9.2.2 Tests sans clause `else` . . . 98
9.2.3 Tests imbriqués . . . 98
9.2.4 Tests à conditions multiples . . . 99
9.2.5 Tests avec initialiseur . . . 100
9.3 Instructions de répétition (boucles) . . . 100
9.3.1 Boucle avec test en début (`while`) . . . 101
9.3.2 Boucle avec test en fin (`do .. while`) . . . 102
9.3.3 Boucle `for` . . . 103
9.3.4 Boucle `for` contre boucle `while` . . . 105
9.4 Sélection (`switch/case/default`) . . . 106
9.4.1 Rôle de l'instruction `break` dans l'instruction `switch` . . . 107
9.4.2 Exemples d'utilisation de `switch` . . . 107
9.4.3 Utiliser `switch` ou `if` ? . . . 108
9.5 Autres instructions contrôlant le flux d'exécution . . . 110
9.5.1 Instruction `break` . . . 110
9.5.2 Instruction `continue` . . . 110
9.5.3 `break`, `continue` et programmation structurée . . . 111
9.5.4 Instruction `goto` . . . 112

10 La fonction `main` **113**
10.1 Arguments des programmes C++ (et C) . . . 113
10.2 Arguments comportant des caractères blancs . . . 115
10.3 Valeur retournée par la fonction `main` . . . 115
10.4 La fonction standard `exit` . . . 115

11 Le préprocesseur **117**
11.1 Qu'est-ce que le préprocesseur ? . . . 117
11.2 Inclure un fichier avec `#include` . . . 118
11.3 Définir une macro avec `#define` . . . 119
11.3.1 Macro sans paramètre . . . 119
11.3.2 Repliement de macro . . . 120
11.3.3 Macro sans valeur . . . 120
11.3.4 Macros prédéfinies par le compilateur . . . 120
11.3.5 Macro avec paramètres . . . 121
11.3.6 Les pièges relatifs aux macros . . . 121
11.3.7 Macros contre fonctions . . . 122
11.4 Suppression de macro avec `#undef` . . . 123
11.5 Compilation conditionnelle . . . 123
11.5.1 Directives `#if`, `#else`, `#elif` et `#endif` . . . 124
11.5.2 Directives `#ifdef`, `#ifndef`, `#elifdef` et `#elifndef` . . . 124
11.6 Autres # directives . . . 125
11.6.1 Affichage d'un message d'erreur fatale avec `#error` . . . 125
11.6.2 Affichage d'un message d'avertissement avec `#warning` . . . 125
11.6.3 Transformation d'un paramètre de macro en chaîne avec `#` . . . 126
11.6.4 Concaténation de paramètres de macro avec `##` . . . 126

12 Déclarations et définitions **127**
12.1 Concepts de déclaration et de définition . . . 127
12.2 Concevoir et analyser des déclarations complexes . . . 128
12.2.1 Principe des déclarations . . . 128
12.2.2 Exemples de déclarations . . . 129
12.2.3 Simplifier une déclaration avec `typedef` ou `using` . . . 131
12.2.4 Analyser une déclaration complexe . . . 132
12.2.5 Exemples d'analyse de déclarations . . . 132
12.2.6 Outils d'assistance aux déclarations . . . 133

13 Visibilité et durée de vie **135**
13.1 Visibilité . . . 135
13.1.1 Entités locales à une fonction . . . 135
13.1.2 Entités globales . . . 136
13.1.3 Entités locales à une unité de compilation . . . 137
13.2 Durée de vie . . . 138
13.2.1 Entités automatiques . . . 138
13.2.2 Entités statiques . . . 139
13.2.3 Entités allouées dynamiquement . . . 139

14 Pointeurs et tableaux **141**
14.1 Arithmétique des pointeurs . . . 141
14.1.1 Addition (ou soustraction) entre un pointeur et un entier . . . 141
14.1.2 Soustraction entre pointeurs . . . 142
14.1.3 Comparaisons entre pointeurs . . . 143
14.1.4 Exemple . . . 143
14.2 Relations entre pointeurs et tableaux . . . 144
14.3 Alors, tableaux et pointeurs sont la même chose ? . . . 145
14.4 Tableaux à indices décalés . . . 146
14.5 Tableaux comme paramètres de fonction . . . 147
14.5.1 Tableau utilisé comme paramètre de sortie . . . 147
14.5.2 Type du paramètre formel lorsque l'argument est un tableau . . . 147
14.5.3 Tableaux de tableaux . . . 148
14.5.4 Tableaux de tableaux de tableaux . . . 150
14.5.5 Tableaux de tableaux de tailles quelconques . . . 151

15 Allocation dynamique de mémoire **153**
15.1 Contexte d'utilisation . . . 153
15.2 Opérateur `new` . . . 153
15.3 Opérateur `new[]` . . . 154
15.4 Opérateur `delete` . . . 154
15.5 Opérateur `delete[]` . . . 155
15.6 Erreurs relatives à l'allocation dynamique . . . 156
15.7 Exemple . . . 157

16 Récursivité **159**
16.1 Coût de la récursivité . . . 160
16.1.1 Coût temporel . . . 160
16.1.2 Coût mémoire . . . 160
16.2 Quand utiliser la récursivité ? . . . 161
16.3 Exemples . . . 162
16.3.1 Implantations récursives efficaces, mais peu intéressantes . . . 162
16.3.2 Implantations récursives peu efficaces . . . 162
16.3.3 Implantations récursives intéressantes . . . 163
16.4 Fonctions à la fois `inline` et récursives . . . 164

III Débuter avec le langage C++ **165**

17 Utiliser C++ sans créer de classe **167**
17.1 La surcharge des fonctions . . . 167
17.1.1 Périmètre d'utilisation de la surcharge des fonctions . . . 168
17.2 Fonctions avec arguments à valeur par défaut . . . 168
17.2.1 Périmètre d'utilisation des paramètres à valeur par défaut . . . 169
17.3 Espaces de noms (`namespace`) . . . 170
17.3.1 Déclaration d'un espace de noms . . . 170
17.3.2 Définition des composants d'un espace de noms . . . 170
17.3.3 Utilisation des composants d'un espace de noms . . . 171
17.3.4 L'espace de noms `std` . . . 172
17.3.5 L'espace de noms global . . . 172
17.3.6 L'espace de noms anonyme . . . 173
17.4 Les références . . . 173
17.4.1 Qu'est-ce qu'une référence ? . . . 173
17.4.2 Références comme paramètres de fonctions . . . 174
17.4.3 Fonctions retournant une référence . . . 175
17.4.4 Références vers constantes . . . 176
17.4.5 Périmètre d'utilisation des références . . . 179
17.4.6 Comment les références sont-elles gérées ? . . . 179
17.4.7 Références contre pointeurs . . . 180
17.4.8 Références à des rvalues . . . 181
17.5 Déclaration par déduction de type avec `auto` . . . 181
17.6 Déclaration par emprunt de type avec `decltype` . . . 181
17.7 Les transtypages à partir de C++11 . . . 181
17.8 Le qualificatif `constexpr` . . . 182
17.9 Le qualificatif `consteval` . . . 184
17.10 Test statique avec `if constexpr` . . . 185
17.11 Vérifications statiques avec `static_assert` . . . 185
17.12 Boucle `for` sur intervalle . . . 187

18 Programmation par objets : les concepts **189**
18.1 Classes et objets . . . 189
18.1.1 Conception basée sur les concepts . . . 189
18.1.2 Classes . . . 189
18.1.3 Objets . . . 189
18.2 Héritage . . . 190
18.2.1 L'héritage en programmation . . . 190
18.2.2 Héritage en cascade . . . 190
18.2.3 Héritage multiple . . . 191
18.2.4 Signification de l'héritage en termes de modélisation . . . 191
18.2.5 Intérêt de l'héritage . . . 191
18.3 Encapsulation . . . 192
18.3.1 Qu'est-ce que l'encapsulation en programmation ? . . . 192
18.3.2 Partie publique . . . 192
18.3.3 Partie privée . . . 192
18.3.4 Partie protégée . . . 192
18.3.5 Intérêt de l'encapsulation . . . 193
18.4 Polymorphisme . . . 193
18.4.1 Le polymorphisme en programmation . . . 193
18.4.2 Intérêt du polymorphisme . . . 194

19 Organisation des classes en C++ **195**
19.1 Des structures aux classes . . . 195
19.2 Encapsulation . . . 195
19.3 Modélisation des données . . . 196
19.4 Modélisation des comportements . . . 196
19.4.1 Notion de méthode et d'objet support . . . 196
19.4.2 Déclaration des méthodes . . . 197
19.4.3 Implantation des méthodes . . . 197
19.5 Organisation du code des classes . . . 197

20 Modéliser un concept : la classe polynôme **201**
20.1 Concept à modéliser . . . 201
20.2 Données caractérisant ce concept . . . 201
20.3 Modéliser les responsabilités : méthodes . . . 202
20.4 Différence entre fonction et méthode . . . 204
20.5 Le pointeur `this` . . . 205
20.6 Méthodes statiques . . . 207
20.7 Méthodes constantes . . . 208
20.8 Fonctions, méthodes et classes amies (`friend`) . . . 210
20.8.1 Fonctions amies . . . 210
20.8.2 Classes amies . . . 211
20.8.3 L'amitié en modélisation . . . 212
20.8.4 Amitié et sécurité . . . 212
20.9 Méthodes `inline` . . . 212

21 Construction et destruction **215**
21.1 Problématique de l'initialisation . . . 215
21.2 Construction . . . 215
21.2.1 Constructeur sans paramètre . . . 215
21.2.2 Constructeur sans paramètre généré par le compilateur . . . 216
21.2.3 Constructeur avec paramètres . . . 217
21.2.4 Constructeur de clonage . . . 219
21.2.5 Construction des objets inclus . . . 223
21.2.6 Syntaxe d'initialisation : `()` ou `{}` ? . . . 225
21.3 Destruction . . . 225

21.3.1 Destructeur . . . 225
21.4 Allocation dynamique de mémoire et construction . . . 226
21.5 Restitution de mémoire et destruction . . . 227

22 Accesseurs **229**
22.1 Qu'est-ce qu'un accesseur ? . . . 229
22.2 Mutateur simple (setter) . . . 229
22.3 Accesseur simple (getter) . . . 229
22.4 Accesseur et attributs calculés . . . 230
22.5 Accesseur et auto-extensibilité . . . 230
22.6 Accesseur pour objet constant . . . 233
22.7 Constance physique, constance logique et attribut `mutable` . . . 234
22.8 `const_cast` contre `mutable` . . . 235
22.9 Périmètre d'utilisation des accesseurs . . . 235

23 Surcharge des opérateurs **237**
23.1 Qu'est-ce que la surcharge des opérateurs ? . . . 237
23.2 Périmètre de la surcharge . . . 237
23.2.1 Opérateurs ne pouvant pas être surchargés . . . 237
23.2.2 Opérateurs `=` et `&` par défaut . . . 238
23.2.3 Sémantique des opérateurs . . . 238
23.3 Techniques de surcharge . . . 238
23.3.1 Implantation d'opérateurs par une méthode . . . 238
23.3.2 Implantation d'opérateurs par une fonction globale . . . 239
23.4 Exemple de surcharge d'opérateurs par méthodes . . . 240
23.4.1 Méthode `operator+` (deux opérandes) . . . 240
23.4.2 Méthode `operator-` (un seul opérande) . . . 241
23.5 Exemple de surcharge d'opérateurs par fonctions globales . . . 241
23.5.1 Fonction globale `operator+` (deux opérandes) . . . 241
23.5.2 Fonction globale `operator-` (un seul opérande) . . . 242
23.6 Exemple d'utilisation des opérateurs . . . 243
23.7 Choisir le mode d'implantation d'un opérateur . . . 243
23.8 Choisir le mécanisme de passage des arguments . . . 243
23.9 Choisir le mécanisme de passage du résultat de l'opération . . . 244
23.10 Les opérateurs importants . . . 245
23.10.1 Opérateur `=` . . . 245
23.10.2 Opérateur de sortie (`<<`) . . . 250
23.10.3 Opérateur d'entrée (`>>`) . . . 252
23.10.4 Opérateurs d'entrée et de sortie : les bonnes pratiques . . . 254
23.11 Autres opérateurs . . . 255
23.11.1 Opérateur `[]` . . . 255
23.11.2 Opérateur `[]` en C++23 . . . 256
23.11.3 Opérateur `*` . . . 257
23.11.4 Opérateur `()` et objets-fonction . . . 259
23.11.5 Opérateurs de comparaison . . . 262

24 Conversions de types **265**
24.1 Conversion implicite . . . 265
24.2 Conversion par constructeur, spécification `explicit` . . . 267
24.3 Conversion par opérateur . . . 268

25 Héritage **273**
25.1 Exemples d'usage . . . 273
25.1.1 Réutilisation d'une classe sans héritage . . . 275
25.1.2 Héritage public . . . 276
25.1.3 Héritage privé . . . 279
25.2 Héritage multiple . . . 283

26 Polymorphisme **285**
26.1 Type statique et type dynamique . . . 285
26.2 Méthodes virtuelles . . . 286
26.3 Périmètre du polymorphisme . . . 287
26.4 Opérateurs polymorphes . . . 287
26.5 Polymorphisme et destruction . . . 288
26.6 Classes abstraites et interfaces . . . 290
26.6.1 Classes abstraites . . . 291
26.6.2 Interfaces . . . 291
26.7 Le polymorphisme en action . . . 292
26.7.1 Représentation d'une expression algébrique . . . 292
26.7.2 La classe `Noeud` . . . 293
26.7.3 Les classes `Noeud1` et `Noeud2` . . . 294
26.7.4 La classe `Constante` . . . 295
26.7.5 Classes `Somme`, `Produit` et `Negation` . . . 296
26.7.6 Création et évaluation d'une expression algébrique . . . 297
26.8 Du bon usage du polymorphisme . . . 298
26.8.1 Remplacement et surcharge : utilisez `override` . . . 298
26.8.2 Polymorphisme et covariance . . . 300

27 Généricité **301**
27.1 Fonctions génériques . . . 301
27.1.1 Écriture d'une fonction générique . . . 301
27.1.2 Utilisation d'une fonction générique . . . 301
27.2 Comment la généricité fonctionne-t-elle en C++ ? . . . 302
27.3 Périmètre de la généricité . . . 303
27.4 Instanciations implicite et explicite . . . 303
27.5 Classes et structures génériques . . . 304
27.5.1 Une classe générique simple . . . 304
27.6 Composants génériques à paramètres entiers . . . 309
27.6.1 Une classe générique à paramètre entier . . . 309
27.7 Paramètres de modèles avec valeurs par défaut . . . 314
27.8 La généricité en action . . . 315
27.8.1 Déclaration de la classe `Polynome` générique . . . 316
27.8.2 Implantation de la classe `Polynome` générique . . . 317
27.8.3 Calcul de la forme polynomiale d'une courbe de Bézier . . . 323

28 Exceptions **327**
28.1 Problématique de gestion des erreurs . . . 327
28.1.1 Un exemple d'erreur de logique . . . 327
28.1.2 Analyse de l'exemple . . . 328
28.1.3 Procédés classiques de traitement des erreurs . . . 328
28.2 Gestion des erreurs basée sur les exceptions . . . 330
28.2.1 Instruction `try` . . . 330
28.2.2 Instruction `throw` . . . 330
28.2.3 Instruction `catch` . . . 331
28.2.4 Exemple . . . 331
28.2.5 Exceptions standards . . . 333
28.2.6 Mode de passage de l'exception à un bloc `catch` . . . 334

28.2.7 Exceptions et héritage . . . 334
28.2.8 Ordre des gestionnaires d'exception . . . 335
28.2.9 Exceptions non interceptées . . . 336
28.2.10 Gestionnaire universel . . . 336
28.2.11 Relancer une exception . . . 337
28.2.12 Exceptions et gestion de ressources : idiome RAII . . . 338

29 Bibliothèque d'entrées-sorties 341
29.1 Structure et composition de la bibliothèque . . . 341
29.1.1 `ios_base` . . . 343
29.1.2 `ios` . . . 343
29.1.3 `ostream` . . . 343
29.1.4 `istream` . . . 343
29.1.5 `iostream` . . . 344
29.1.6 `ostringstream` . . . 344
29.1.7 `ofstream` . . . 344
29.1.8 `istringstream` . . . 344
29.1.9 `ifstream` . . . 344
29.1.10 `stringstream` . . . 344
29.1.11 `fstream` . . . 344
29.1.12 Flux standards préouverts . . . 344
29.2 Les différents états des flux . . . 345
29.2.1 `fail`, `bad`, `eof` et `good` . . . 345
29.2.2 `operator void *` et `operator bool` . . . 347
29.2.3 Modifier l'état d'un flux avec `clear` . . . 347
29.3 Les différents modes d'ouverture des flux . . . 349
29.4 `istream` . . . 349
29.4.1 Lecture formatée . . . 350
29.4.2 Lecture de caractères . . . 350
29.4.3 Positionnement dans un flux d'entrée . . . 352
29.5 `ifstream` . . . 354
29.5.1 Construction et destruction . . . 354
29.5.2 `rdbuf` . . . 354
29.5.3 `open` . . . 354
29.5.4 `close` . . . 354
29.5.5 `is_open` . . . 354
29.5.6 Exemple . . . 355
29.6 `istringstream` . . . 355
29.6.1 Construction et destruction . . . 355
29.6.2 `rdbuf` . . . 355
29.6.3 `str` . . . 356
29.6.4 Exemple . . . 356
29.7 `ostream` . . . 357
29.7.1 Écritures non formatées . . . 357
29.7.2 Écritures formatées . . . 357
29.7.3 Positionnement dans un flux de sortie . . . 357
29.8 `ofstream` . . . 358
29.8.1 Construction et destruction . . . 358
29.8.2 `rdbuf` . . . 358
29.8.3 `open` . . . 358
29.8.4 `close` . . . 358
29.8.5 `is_open` . . . 358
29.9 `ostringstream` . . . 358
29.9.1 Construction et destruction . . . 358
29.9.2 `rdbuf` . . . 358
29.9.3 `str` . . . 359

29.10 `iostream`, `fstream` et `stringstream` . . . 359
29.11 Manipulateurs d'entrées-sorties . . . 360
29.11.1 Rôle des manipulateurs . . . 360
29.11.2 Fichiers d'en-tête à inclure . . . 360
29.11.3 Manipulateurs déclarés dans `<ios>` . . . 360
29.11.4 Manipulateurs déclarés dans `<istream>` . . . 362
29.11.5 Manipulateurs déclarés dans `<ostream>` . . . 363
29.11.6 Manipulateurs déclarés dans `<iomanip>` . . . 363
29.12 Créer ses propres manipulateurs . . . 366
29.12.1 Manipulateurs sans paramètre . . . 366
29.12.2 Manipulateurs avec paramètres . . . 367
29.13 Bibliothèque `format` . . . 368
29.13.1 Fonction `format` . . . 369
29.13.2 Fonction `format_to` . . . 374
29.13.3 Fonction `print` . . . 374
29.14 Extension de la bibliothèque `format` . . . 375
29.14.1 Prérequis . . . 375
29.14.2 API d'extension de la bibliothèque `format` . . . 375
29.14.3 Formatage du type `Monome` . . . 376

30 Expressions lambda (lambda-fonctions) 379
30.1 Qu'est-ce qu'une expression lambda ? . . . 380
30.2 Expressions lambda « nommées » . . . 381
30.3 Expressions lambda génériques . . . 382
30.4 Type de retour d'une expression lambda . . . 383
30.5 Expression lambda avec paramètres par défaut . . . 383
30.6 Capture de contexte dans une expression lambda . . . 384
30.6.1 Captures par valeur . . . 384
30.6.2 Captures par référence . . . 386
30.6.3 Captures mixtes . . . 387
30.7 Exemples . . . 387
30.7.1 Transmission d'un algorithme . . . 388
30.7.2 Curryfication . . . 388

31 Bibliothèque générique standard (STL) 391
31.1 Qu'est-ce que la STL ? . . . 391
31.2 Introduction aux conteneurs . . . 391
31.3 Introduction aux itérateurs . . . 392
31.3.1 Les différentes catégories d'itérateurs . . . 395
31.4 Propriétés des conteneurs . . . 396

32 Conteneurs de la STL 397
32.1 Le conteneur `vector` . . . 397
32.1.1 Description générale . . . 397
32.1.2 Modèle d'extension . . . 397
32.1.3 Construction d'un `vector` . . . 397
32.1.4 Insertion d'éléments . . . 398
32.1.5 Accès aux éléments . . . 399
32.1.6 Propriétés . . . 399
32.1.7 Périmètre d'utilisation de `vector` . . . 400
32.1.8 Exemple . . . 400
32.2 Le conteneur `deque` . . . 404
32.2.1 Description générale . . . 404
32.2.2 Modèle d'extension . . . 404
32.2.3 Construction d'un `deque` . . . 404
32.2.4 Insertion d'éléments . . . 405

32.2.5 Accès aux éléments . . . 406
32.2.6 Propriétés . . . 406
32.2.7 Périmètre d'utilisation . . . 407
32.2.8 Exemple . . . 407
32.3 Le conteneur `array` . . . 409
32.3.1 Description générale . . . 409
32.3.2 Modèle d'extension . . . 409
32.3.3 Construction . . . 409
32.3.4 Insertion d'éléments . . . 410
32.3.5 Accès aux éléments . . . 410
32.3.6 Propriétés . . . 410
32.3.7 Périmètre d'utilisation . . . 410
32.3.8 Exemple . . . 411
32.4 Le conteneur `string` (`basic_string<char>`) . . . 411
32.4.1 Description générale . . . 411
32.4.2 Modèle d'extension . . . 412
32.4.3 Construction . . . 412
32.4.4 Accès aux éléments . . . 413
32.4.5 Propriétés . . . 413
32.4.6 Périmètre d'utilisation . . . 414
32.4.7 Constantes de type `string` . . . 414
32.4.8 Exemple . . . 415
32.5 Les conteneurs `forward_list` et `list` . . . 417
32.5.1 Description générale . . . 417
32.5.2 Modèle d'extension . . . 418
32.5.3 Construction . . . 419
32.5.4 Accès aux éléments . . . 419
32.5.5 Insertion et suppression d'éléments . . . 420
32.5.6 Algorithmes intégrés . . . 420
32.5.7 Propriétés . . . 420
32.5.8 Périmètre d'utilisation . . . 421
32.5.9 Exemple . . . 421
32.6 Les conteneurs `map` et `multimap` . . . 422
32.6.1 Description générale . . . 422
32.6.2 Modèle d'extension . . . 424
32.6.3 Création . . . 425
32.6.4 Insertion d'éléments . . . 425
32.6.5 Accès aux éléments . . . 426
32.6.6 Propriétés . . . 426
32.6.7 Périmètre d'utilisation . . . 426
32.6.8 Exemple . . . 427
32.7 Les conteneurs `unordered_map` et `unordered_multimap` . . . 431
32.7.1 Description générale . . . 432
32.7.2 Différence entre `unordered_map` et `unordered_multimap` . . . 432
32.7.3 Modèle d'extension . . . 432
32.7.4 Construction . . . 432
32.7.5 Insertion d'éléments . . . 434
32.7.6 Accès aux éléments . . . 435
32.7.7 Propriétés . . . 435
32.7.8 Périmètre d'utilisation . . . 435
32.7.9 Exemple d'utilisation . . . 435
32.8 Les conteneurs `flat_map` et `flat_multimap` (C++23) . . . 439
32.8.1 Description générale . . . 439
32.8.2 Modèle d'extension et utilisation . . . 440
32.8.3 Propriétés . . . 440

32.8.4 Périmètre d'utilisation . . . 441
32.8.5 Exemple d'utilisation . . . 441
32.9 Les conteneurs `set` et `multiset` . . . 441
32.9.1 Description générale . . . 442
32.10 Les conteneurs `unordered_set` et `unordered_multiset` . . . 442
32.10.1 Description générale . . . 442
32.11 Les conteneurs `flat_set` et `flat_multiset` . . . 442
32.11.1 Description générale . . . 442
32.12 Le conteneur `stack` . . . 443
32.12.1 Description générale . . . 443
32.12.2 Modèle d'extension . . . 443
32.12.3 Construction . . . 444
32.12.4 Insertion d'éléments . . . 444
32.12.5 Accès aux éléments . . . 444
32.12.6 Propriétés . . . 444
32.12.7 Périmètre d'utilisation . . . 444
32.12.8 Exemple d'utilisation . . . 444
32.13 Le conteneur `queue` . . . 446
32.13.1 Description générale . . . 447
32.13.2 Modèle d'extension . . . 447
32.13.3 Construction . . . 447
32.13.4 Insertion d'éléments . . . 447
32.13.5 Accès aux éléments . . . 447
32.13.6 Propriétés . . . 448
32.13.7 Périmètre d'utilisation . . . 448
32.14 Le conteneur `priority_queue` . . . 448
32.14.1 Description générale . . . 448
32.14.2 Modèle d'extension . . . 449
32.14.3 Construction . . . 449
32.14.4 Insertion d'éléments . . . 449
32.14.5 Accès aux éléments . . . 449
32.14.6 Propriétés . . . 449
32.14.7 Périmètre d'utilisation . . . 450
32.14.8 Exemple d'utilisation . . . 450
32.15 Le mandataire `span` . . . 452
32.15.1 Description générale . . . 452
32.15.2 Construction . . . 452
32.15.3 Accès aux éléments . . . 452
32.15.4 Propriétés . . . 453
32.15.5 Périmètre d'utilisation . . . 453
32.15.6 Exemple d'utilisation . . . 453
32.16 Le mandataire `string_view` . . . 455
32.16.1 Description générale . . . 455
32.16.2 Construction . . . 455
32.16.3 Manipulation d'un `string_view` . . . 455
32.16.4 Périmètre d'utilisation . . . 456
32.16.5 Exemple d'utilisation . . . 457
32.17 La bibliothèque `ranges` . . . 458
32.17.1 Les différentes catégories de `ranges` . . . 458
32.17.2 Les vues . . . 459
32.18 Composition de vues . . . 460
32.18.1 Fabriques de ranges . . . 461

33 Algorithmes de la STL **463**
33.1 Introduction aux algorithmes . . . 463
33.1.1 Algorithmes et itérateurs . . . 463
33.1.2 Les différentes catégories d'itérateurs . . . 464
33.1.3 Les adaptateurs . . . 464
33.2 Algorithmes non modifiants . . . 465
33.2.1 `all_of`, `any_of`, `none_of` . . . 465
33.2.2 `for_each`, `for_each_n` . . . 465
33.2.3 `count`, `count_if` . . . 465
33.2.4 `mismatch` . . . 466
33.2.5 `find`, `find_if`, `find_if_not` . . . 466
33.2.6 `find_end` . . . 466
33.2.7 `find_first_of` . . . 466
33.2.8 `adjacent_find` . . . 466
33.2.9 `search` . . . 466
33.2.10 `search_n` . . . 466
33.2.11 Exemples . . . 467
33.3 Algorithmes modifiants . . . 469
33.3.1 `copy`, `copy_if`, `copy_n`, `copy_backward` . . . 469
33.3.2 `move`, `move_backward` . . . 470
33.3.3 `fill`, `fill_n` . . . 470
33.3.4 `transform` . . . 470
33.3.5 `generate`, `generate_n` . . . 470
33.3.6 `remove`, `remove_if` . . . 470
33.3.7 `remove_copy`, `remove_copy_if` . . . 471
33.3.8 `replace`, `replace_if` . . . 471
33.3.9 `replace_copy`, `replace_copy_if` . . . 471
33.3.10 `swap` . . . 471
33.3.11 `iter_swap` . . . 471
33.3.12 `swap_ranges` . . . 471
33.3.13 `reverse`, `reverse_copy` . . . 472
33.3.14 `rotate`, `rotate_copy` . . . 472
33.3.15 `shuffle`, `random_shuffle` . . . 472
33.3.16 `sample` . . . 472
33.3.17 `unique` , `unique_copy` . . . 472
33.3.18 Exemples . . . 473
33.4 Algorithmes de partitionnement et de tri . . . 477
33.4.1 `partition` . . . 477
33.4.2 `stable_partition` . . . 477
33.4.3 `partition_point` . . . 477
33.4.4 `is_sorted` . . . 477
33.4.5 `is_sorted_until` . . . 477
33.4.6 `sort` . . . 477
33.4.7 `stable_sort` . . . 478
33.4.8 `partial_sort` . . . 478
33.4.9 `partial_sort_copy` . . . 478
33.4.10 `nth_element` . . . 478
33.4.11 Exemples utilisant tris et partitionnements . . . 478
33.5 Algorithmes sur intervalles triés . . . 481
33.5.1 `lower_bound`, `upper_bound` . . . 481
33.5.2 `binary_search` . . . 481
33.5.3 `equal_range` . . . 481
33.5.4 Exemples . . . 481
33.6 Algorithmes ensemblistes . . . 482
33.6.1 `merge` . . . 483

33.6.2 `inplace_merge` . . . 483
33.6.3 `includes` . . . 483
33.6.4 `set_difference` . . . 483
33.6.5 `set_symmetric_difference` . . . 483
33.6.6 `set_intersection` . . . 484
33.6.7 `set_union` . . . 484
33.6.8 Exemples . . . 484
33.7 Algorithmes sur les tas (*heaps*) . . . 486
33.7.1 Définition . . . 486
33.7.2 `make_heap` . . . 487
33.7.3 `is_heap` . . . 487
33.7.4 `is_heap_until` . . . 487
33.7.5 `push_heap` . . . 487
33.7.6 `pop_heap` . . . 487
33.7.7 `sort_heap` . . . 487
33.7.8 Exemples . . . 487

34 Type `initializer_list` 489
34.1 Création d'un `initializer_list` . . . 489
34.2 Interface du modèle `initializer_list` . . . 489
34.3 Exemple . . . 490

35 Liaison structurée (*Structured binding*) 491
35.1 Collections autorisant les liaisons structurées . . . 492
35.2 Usages typiques . . . 494

36 Pointeurs sur membres 495
36.1 Introduction . . . 495
36.2 Exemple . . . 496

37 Pratiquez le « C++ facile » 501
37.1 Limitez l'usage des pointeurs . . . 501
37.2 Passez les paramètres par valeur ou par référence . . . 501
37.3 N'utilisez pas le système d'entrées-sorties du C . . . 502
37.4 N'utilisez pas les tableaux . . . 502
37.5 N'utilisez pas les chaînes de caractères du langage C . . . 503
37.6 Limitez l'usage du préprocesseur . . . 503
37.7 Ne gérez pas la mémoire vous-même, utilisez la STL . . . 504
37.8 Utilisez les exceptions . . . 504
37.9 Pratiquez l'idiome RAII . . . 504

IV Approfondir le langage C++ 505

38 Construction et destruction 507
38.1 Les différentes sortes d'objets . . . 507
38.2 Construction et destruction d'objets automatiques . . . 508
38.3 Construction et destruction d'objets statiques . . . 508
38.4 Construction et destruction d'objets dynamiques . . . 508
38.5 Construction et destruction d'objets membres ou de base . . . 508
38.6 Construction et destruction d'éléments d'un tableau . . . 509
38.7 Construction et exception . . . 510
38.8 Syntaxe d'initialisation : utiliser `{}` ou `()` ? . . . 511

39 Allocation dynamique **513**
39.1 Personnaliser `new`, `new[]`, `delete` et `delete[]` . . . 513
39.1.1 Opérateur `new` . . . 513
39.1.2 Opérateur `delete` . . . 514
39.1.3 Opérateur `new[]` . . . 514
39.1.4 Opérateur `delete[]` . . . 515
39.1.5 Exemple . . . 515
39.2 Opérateur `new` avec placement . . . 517

40 Opérateurs **521**
40.1 Opérateur `=` . . . 521
40.1.1 Implantation de l'opérateur `=` par l'idiome swap . . . 521
40.1.2 Gestion des auto-affectations . . . 523
40.1.3 L'opérateur `=` retourne une lvalue en C++ . . . 523
40.2 Opérateurs `++` et `--` . . . 523
40.2.1 Que retournent `++` et `--`? . . . 523
40.2.2 Distinction entre les formes préfixées et postfixées . . . 524
40.2.3 Exemple de surcharge de l'opérateur `++` . . . 525
40.2.4 Un opérateur `++` postfixé gratuit? . . . 527
40.3 Opérateur `->` . . . 528
40.3.1 Fonctionnement de l'opérateur `->` . . . 528
40.3.2 Un mandataire basé sur l'opérateur `->` . . . 528
40.3.3 Un pointeur intelligent basé sur l'opérateur `->` . . . 530
40.4 Opérateur `""` . . . 532
40.4.1 Principes de mise en œuvre . . . 533
40.4.2 Exemple : littéral de type `short` . . . 534
40.4.3 Exemple : littéral exprimé dans une unité particulière . . . 534
40.4.4 Exemple : internationalisation des chaînes de caractères . . . 535
40.4.5 Littéraux définis par l'utilisateur et `constexpr` . . . 536
40.5 Opérateurs statiques (C++23) . . . 537
40.6 Opérateur de comparaison à trois voies `<=>` (C++20) . . . 538
40.6.1 Introduction . . . 538
40.6.2 Opérateur `<=>` par défaut . . . 538
40.6.3 Opérateur `<=>` fourni par l'utilisateur . . . 540
40.6.4 Type du résultat des comparaisons . . . 540
40.6.5 Conversions entre résultats de comparaisons . . . 541
40.6.6 Comparaisons de résultats de comparaisons avec l'entier `0` . . . 541
40.6.7 Cas particulier de l'opérateur `==` . . . 542
40.6.8 Exemple . . . 542
40.7 Repliement d'opérateur . . . 543
40.8 Les quatre manières de replier une expression . . . 544
40.9 Quelques cas d'utilisation . . . 545
40.9.1 Opérateurs non commutatifs . . . 545
40.9.2 Expressions repliées complexes . . . 546
40.9.3 Expressions repliées à effet de bord . . . 546
40.9.4 Ordre d'évaluation des opérandes . . . 547
40.10 Opérateurs nommés . . . 548
40.10.1 Création d'un opérateur nommé . . . 549

41 Transtypages **551**
41.1 `static_cast` . . . 551
41.2 `const_cast` . . . 552
41.3 `reinterpret_cast` . . . 552
41.4 `dynamic_cast` . . . 554
41.4.1 Downcasting avec `dynamic_cast` . . . 557
41.4.2 Sidecasting avec `dynamic_cast` . . . 558

42 Identification de type à l'exécution (RTTI) 559
42.1 L'opérateur `typeid` . . . 559
42.2 La classe `type_info` . . . 560
42.2.1 Comparaisons de `type_info` . . . 560
42.2.2 Obtenir le nom du type représenté par un `type_info` . . . 561
42.2.3 Classer des `type_info` . . . 563
42.2.4 Tables de hachage contenant des `type_info` . . . 564

43 Support du polymorphisme 565
43.1 Principe général . . . 565
43.2 Organisation de la table des méthodes virtuelles . . . 567
43.3 Invocation des méthodes virtuelles . . . 568
43.4 Obtention du type d'un objet polymorphe . . . 569
43.5 Conclusion . . . 570

44 Héritage 571
44.1 Héritage, encapsulation et pointeurs . . . 571
44.1.1 Affectation de l'adresse d'un objet dérivé à un pointeur sur base, en cas d'héritage public . . . 571
44.1.2 Affectation de l'adresse d'un objet dérivé à un pointeur sur base, en cas d'héritage privé . . . 572
44.1.3 Affectation d'un pointeur sur objet de base à un pointeur sur objet dérivé. . . . 572
44.2 Héritage et passage de paramètres . . . 574
44.3 Héritage et affectation . . . 575
44.4 Affectation polymorphe . . . 577
44.5 Héritage multiple . . . 580
44.5.1 Alternatives à l'héritage multiple . . . 580
44.5.2 Héritage multiple et polymorphisme . . . 581
44.5.3 Héritage multiple et ambiguïtés . . . 581
44.5.4 Héritage en diamant et héritage virtuel . . . 583
44.5.5 Exemple d'héritage virtuel en diamant . . . 585
44.6 Héritage dynamique . . . 589
44.6.1 Le pattern décorateur . . . 590
44.6.2 Pattern décorateur et downcasting . . . 593
44.7 Construction et destruction en cas d'héritage multiple ou virtuel . . . 598

45 Généricité 599
45.1 Modèles d'alias (alias template) . . . 599
45.2 Spécialisation totale des modèles . . . 600
45.3 Spécialisations partielles des modèles . . . 601
45.4 Modèles de paramètres de modèles . . . 604
45.5 Guides de déduction . . . 607
45.5.1 Guides de déduction simples . . . 608
45.5.2 Modèles de guides de déduction . . . 608
45.6 Modèles variadiques (variadic templates) . . . 610
45.6.1 L'opérateur `sizeof...` . . . 610
45.6.2 Modèle variadique de fonction . . . 611
45.6.3 Modèle variadique de classe . . . 615
45.6.4 Les modèles variadiques sont-ils importants en C++? . . . 618

46 Déduction de type 619
46.1 Déduction de type dans les modèles . . . 619
46.1.1 Paramètre formel déclaré comme une référence . . . 619
46.1.2 Paramètre formel déclaré comme un pointeur . . . 620
46.1.3 Paramètre passé par valeur . . . 621
46.1.4 Paramètre réel de type tableau . . . 621

46.2 Déduction de type avec `auto` . . . 622
46.2.1 Références explicitement déclarées avec `auto` . . . 622
46.2.2 Pointeurs explicitement déclarés avec `auto` . . . 622
46.2.3 Entités déclarées avec `auto` seul . . . 623
46.3 Différence entre `auto` et `decltype` . . . 623

47 Références aux rvalues 625
47.1 rvalue et lvalue : quelques rappels . . . 625
47.2 Passage d'une rvalue en paramètre . . . 626
47.3 Références aux rvalues . . . 627
47.4 Transmissions par déplacement . . . 628
47.5 Conversion en rvalue avec `std::move` . . . 630
47.6 Méthodes à objet support déplaçable . . . 632

48 Transmissions parfaites 635
48.1 Réduction des références et références universelles . . . 635
48.2 Transtypage conditionnel avec `std::forward` . . . 638
48.3 Exemple . . . 639
48.3.1 Transmission parfaite d'un seul paramètre . . . 640
48.3.2 Transmission parfaite de plusieurs paramètres . . . 641
48.4 Les transmissions parfaites dans la STL . . . 643

49 Pointeurs intelligents (smart pointers) 645
49.1 Introduction . . . 645
49.2 Pourquoi utiliser des pointeurs intelligents ? . . . 645
49.3 `unique_ptr` . . . 645
49.3.1 Création d'un `unique_ptr` . . . 646
49.3.2 Accès aux données pointées par un `unique_ptr` . . . 648
49.3.3 Affectation entre `unique_ptr` . . . 648
49.3.4 Faire pointer un `unique_ptr` sur un autre objet . . . 649
49.3.5 Permuter des `unique_ptr` . . . 649
49.3.6 `unique_ptr` gérant un tableau . . . 649
49.3.7 `unique_ptr` avec effaceur spécifique . . . 650
49.3.8 La fonction `make_unique` . . . 652
49.3.9 `unique_ptr` et transmissions par déplacement . . . 653
49.4 `shared_ptr` . . . 653
49.4.1 Principe de fonctionnement des `shared_ptr` . . . 654
49.4.2 Création d'un `shared_ptr` . . . 655
49.4.3 Faire pointer un `shared_ptr` sur un autre objet . . . 656
49.4.4 Accès aux données pointées par un `shared_ptr` . . . 657
49.4.5 Affectation entre `shared_ptr` . . . 657
49.4.6 Les `shared_ptr` avec effaceurs spécifiques . . . 657
49.4.7 La fonction `make_shared` . . . 658
49.4.8 Les `shared_ptr` et la programmation concurrente . . . 658
49.4.9 Exemple d'utilisation de `shared_ptr` . . . 659
49.5 `weak_ptr` . . . 660
49.5.1 Création d'un `weak_ptr` . . . 662
49.5.2 Dissociation d'un `weak_ptr` et de l'objet pointé . . . 663
49.5.3 Accès aux données pointées par un `weak_ptr` grâce à `lock` . . . 663
49.5.4 Affectation entre `weak_ptr`, `shared_ptr` et `unique_ptr` . . . 665
49.5.5 Un exemple d'utilisation de `weak_ptr` . . . 665

50 Expressions lambda et closures **671**
50.1 Implantation des expressions lambda . . . 671
50.2 Type d'une closure . . . 671
50.3 Rôle des closures . . . 672
50.3.1 Capture de l'environnement . . . 672
50.3.2 Exécution des closures . . . 672
50.4 Capture généralisée . . . 673
50.5 Capture par déplacement . . . 674
50.6 Expressions lambda et références pendantes . . . 675
50.6.1 Capture par valeur . . . 675
50.7 Capture par valeur de l'objet support . . . 677
50.8 Expressions lambda génériques et transmissions parfaites . . . 678

51 Exceptions **681**
51.1 Gestion des exceptions non interceptées . . . 681
51.2 Spécifications d'exceptions avant C++11 . . . 682
51.3 Spécifications d'exceptions à partir de C++11 . . . 683
51.3.1 Le mot-clé `noexcept` . . . 683
51.3.2 Garantie offerte par `noexcept` . . . 683
51.3.3 `noexcept(`*prédicat_constant*`)` . . . 683
51.3.4 Exemple utilisant le mot-clé `noexcept` . . . 684
51.3.5 Opérateur `noexcept` . . . 685
51.3.6 Exemple utilisant l'opérateur `noexcept` . . . 686
51.4 Méthodes générées automatiquement et exceptions . . . 686
51.5 Écrire du code sûr vis-à-vis des exceptions . . . 689
51.5.1 Quelles sont les sources d'insécurité ? . . . 689
51.5.2 Sur quoi peut-on compter pour réaliser des composants sûrs ? . . . 691
51.5.3 Sûreté vis-à-vis des exceptions de la classe `Polynome` . . . 691

52 Itérateurs **699**
52.1 Un itérateur pour le type intervalle . . . 699
52.2 Propriétés des différentes catégories d'itérateurs . . . 700
52.3 Conception de la classe `IRange` et de son itérateur . . . 700
52.4 Rendre les itérateurs compatibles avec l'écosystème C++ . . . 704

53 Programmation concurrente et threads **707**
53.1 Qu'est-ce qu'un thread ? . . . 707
53.2 Premiers pas avec les threads . . . 708
53.3 Paramètres de construction d'un thread . . . 711
53.4 Les différentes façons d'implanter un thread . . . 712
53.4.1 Thread implanté par un objet fonction . . . 713
53.4.2 Thread implanté par une lambda . . . 714
53.5 Threads et exclusion mutuelle . . . 714
53.5.1 Sections critiques . . . 714
53.5.2 Gestion des accès concurrents . . . 715
53.5.3 Sections critiques et exceptions . . . 717
53.5.4 Variables atomiques (modèle `atomic<T>`) . . . 718
53.6 Variables de condition et synchronisation . . . 719
53.6.1 Principe de fonctionnement . . . 719
53.6.2 Exemple de synchronisation de tâches asynchrones . . . 720
53.7 `jthread` . . . 722
53.7.1 Auto-joignabilité . . . 722
53.7.2 Arrêt prématuré d'un `jthread` . . . 723

54 Programmation concurrente et tâches **727**
54.1 Création d'une tâche avec `async` . . . 727
54.2 Détection de la terminaison d'une tâche . . . 728
54.3 Tâches et exceptions . . . 729
54.4 Fonctionnement des tâches (`promise`, `future`) . . . 730
54.4.1 Mécanisme de transmission du résultat d'une tâche . . . 730

55 Objets exécutables asynchrones **733**
55.1 Introduction aux `packaged_task` . . . 733
55.2 Le modèle `function` . . . 733
55.3 Le modèle `packaged_task` . . . 734
55.4 `packaged_task` et exécution concurrente . . . 735

56 Métaprogrammation **737**
56.1 Introduction . . . 737
56.1.1 Principe . . . 737
56.1.2 Métaprogrammation et `constexpr` . . . 738
56.2 SFINAE . . . 739
56.2.1 Principe . . . 739
56.2.2 Périmètre d'application de SFINAE . . . 740
56.2.3 Liste des erreurs SFINAE . . . 741
56.2.4 Exemple : sélection d'un modèle par le type d'un argument . . . 741
56.2.5 Exemple : sélection d'un modèle par une valeur entière . . . 744
56.3 Introspection . . . 746
56.3.1 Introspection statique avec SFINAE . . . 746
56.3.2 Introspection dynamique . . . 750
56.4 Métaprogrammation et bibliothèque standard . . . 751

57 Concepts **753**
57.1 Utilisation d'un concept prédéfini . . . 753
57.2 Création d'un concept . . . 754
57.2.1 Concepts et SFINAE . . . 756
57.2.2 Sélectionner un modèle de fonction selon le type de son argument . . . 756
57.2.3 Concept basé sur un paramètre entier . . . 758
57.2.4 Vérification des propriétés d'un composant . . . 759

58 Modules **761**
58.1 Qu'est-ce qu'un module ? . . . 762
58.2 Fichiers décrivant les modules . . . 763
58.3 Création d'un module simple . . . 763
58.4 Modules et bibliothèque standard . . . 765
58.5 Utilisation d'entités non importables . . . 766
58.6 Sous-modules . . . 768
58.7 Partitions . . . 770
58.8 Modularisation d'un composant préexistant . . . 772
58.9 Exportation et espaces de noms . . . 773
58.10 Conception des modules . . . 774
58.10.1 Minimiser les exportations . . . 774
58.10.2 Choisir la taille des modules . . . 775
58.10.3 Organiser le code des modules . . . 775

59 Objet support explicite (C++23) **777**
59.1 Introduction . . . 777
59.2 Création de méthodes à objet support explicite . . . 777
59.2.1 Propriétés et nature des méthodes OSE . . . 778
59.3 Applications des méthodes OSE . . . 778
59.3.1 Objet support passé par valeur . . . 779
59.3.2 Lambda fonctions récursives . . . 779
59.3.3 Unification (ou déduplication) dans un cas simple . . . 780
59.3.4 Autres cas d'unification . . . 781
59.3.5 Polymorphisme statique . . . 785

A Annexes **789**
A.1 Notions de base en programmation . . . 789
A.1.1 Qu'est-ce qu'un ordinateur ? . . . 789
A.1.2 Langages de programmation . . . 797
A.1.3 Les environnements de développement . . . 804
A.2 Introduction aux commandes de compilation (`g++`/`clang++`) . . . 805
A.2.1 Compilation d'un seul fichier . . . 806
A.2.2 Compilation de plusieurs fichiers . . . 806
A.2.3 Édition de liens . . . 806
A.2.4 Compilation et édition de liens en une seule commande . . . 806
A.2.5 Compilation et édition de liens avec une bibliothèque . . . 806
A.3 Construction des applications utilisant les modules . . . 807
A.3.1 Création des unités d'en-tête (header units) . . . 807
A.3.2 Compilation des déclarations de modules . . . 807
A.3.3 Compilation des implantations de modules . . . 808
A.3.4 Construction de l'exécutable . . . 808
A.4 Interopérabilité entre des codes C et C++ . . . 809
A.4.1 Bibliothèques C standardisées par la norme C++ . . . 809
A.4.2 Invocation de fonctions C à partir de code C++ . . . 809
A.4.3 Création de fonctions C à partir de code C++ . . . 810

B Bibliographie **811**
B.1 Livres . . . 811
B.2 Ressources Web . . . 812
B.3 « Intelligence » artificielle générative . . . 812

Première partie

Préambule

1 | À propos de ce livre

1.1 À qui s'adresse-t-il ?

Comme dans sa première édition, ce livre a été conçu pour accompagner les étudiants d'université ou d'école d'ingénieurs dès le début de leur cursus. Mais il conviendra aussi à toute personne désirant compléter ou actualiser ses compétences de programmeur en C++. Par le traitement de sujets de difficulté croissante, il se propose de vous guider de manière progressive vers la maîtrise de ce langage très puissant qu'est C++, dans sa plus récente version normalisée : C++23.

1.2 Que contient-il ?

Cet ouvrage est consacré au langage C++. Toutefois, ce dernier est construit sur le langage C, et la relation qui existe entre ces langages est davantage qu'une simple filiation : C++ est un *surensemble* de C. Le choix qui a été fait dans cette édition est de traiter l'héritage du langage C dans une première partie intitulée « *Le socle de C++* ». On y présente uniquement les éléments du langage C repris par C++, en excluant ceux qui sont inutiles, voire nuisibles.

Il est impossible de décrire un langage de programmation d'une manière à la fois détaillée et accessible aux débutants. Il est en effet fréquent qu'une description approfondie fasse référence à des informations ou à des concepts qui n'ont pas encore été présentés. Pour cette raison, la présentation du langage C++ a été scindée en deux parties.

La première, « *Débuter avec le langage C++* », contient une description du langage largement suffisante pour écrire des programmes intéressants. Elle débute par une description de ce qu'est la *programmation par objets*, et des concepts associés à ce type d'approche. L'objectif de cette présentation est de montrer que la programmation par objets ne s'appuie pas uniquement sur l'usage de « boîtes à outils » logicielles : c'est aussi une démarche d'analyse des problèmes permettant de concevoir l'architecture de leurs solutions.

Une introduction progressive des fonctionnalités de base de C++ vient ensuite. Elle s'appuie sur des exemples d'abord rudimentaires, puis plus élaborés au fil des chapitres. « Progressive » signifie ici que les éléments du langage sont présentés dans un ordre logique, en commençant par les plus primitifs. Chaque sujet abordé s'appuie ainsi sur les sujets précédents afin de minimiser les risques d'incompréhension.

La seconde partie intitulée « *Approfondir le langage C++* » traite de sujets un peu plus ardus, et même parfois difficiles si l'on s'intéresse aux aspects les plus avancés. Il faut s'attendre à ce que certaines de ces pages nécessitent plusieurs lectures pour être assimilées. Mais ne vous inquiétez pas, car vous n'êtes pas obligés de tout savoir sur le langage C++ pour écrire des programmes intéressants et très performants. Certains chapitres portent le même nom dans la première et la seconde partie, car traitent du même sujet, mais à des niveaux de détail différents.

1.3 Les exemples

Apprendre un langage de programmation peut être frustrant si chaque point abordé n'est pas illustré par un exemple immédiatement utilisable. Cependant, même simples, les premiers exemples sont difficiles à introduire car il n'est pas possible de présenter simultanément les connaissances requises pour les comprendre. On est donc parfois amené à présenter du code comportant des

instructions ou utilisant des concepts qui n'ont pas encore été introduits. Dans ce cas, ou lorsque le sujet est difficile, le texte comporte de très nombreux renvois (comportant numéros de sections et de pages) permettant d'approfondir les points évoqués.

Les exemples courts ont l'avantage d'illustrer de manière facilement compréhensible les fonctionnalités présentées. Toutefois, ils ne permettent généralement pas d'en démontrer l'intérêt concret. C'est pour cela que ce livre contient beaucoup d'exemples plus ambitieux qui replacent le sujet dans un contexte pratique. Ceux-ci sont plus difficiles à comprendre, car ils tiennent compte de détails qui, bien qu'ils débordent parfois du cadre du sujet traité, rendent le code réaliste.

Il existe dans ce livre trois sortes d'exemples, tous imprimés avec une police à chasse fixe, mais avec une présentation spécifique :

(a) Un *fragment de code* est un court extrait destiné à illustrer un point précis. Les fragments ne sont pas utilisables en l'état, car ils sont incomplets : il leur manque généralement un contexte, par exemple des déclarations ou initialisations de variables. Voici un exemple de fragment de code :

```
/* Schéma d'HORNER */
y = ((a * x) + b ) * x + c  ;
```

Dans tous les exemples, les commentaires avec points de suspension tels que :

```
// ...
/* ... */
// ... suite de l'exemple précédent
```

indiquent l'emplacement d'un code qui n'est pas en relation directe avec le sujet, et qui a été volontairement omis pour simplifier la présentation.

(b) Les *fonctions* (ou les *classes*) constituent des outils presque complets, et sont donc généralement utilisables dans un programme par simple copier-coller. Pour des raisons de simplicité et de brièveté, certains éléments sont cependant omis dans les exemples présentés : c'est généralement le cas des inclusions de fichiers, pourtant nécessaires à la compilation de la fonction. Par exemple, une fonction qui réalise des entrées-sorties en langage C++ devrait être précédée de l'inclusion du fichier d'en-tête `<iostream>`, mais ce n'est pas forcément le cas dans ce livre afin d'éviter de le rendre encore plus volumineux.

Voici un exemple de fonction :

```
1  // ...
2  /*
3    Calcule la valeur d'un polynôme du second degré,
4    de coefficients a,b,c au point x
5  */
6  double poly2(double x, double a, double b, double c)
7  {
8    double y ;
9    // Schéma d'HORNER
10   y = ((a * x) + b ) * x + c  ;
11   return y ;
12 }
13 // ...
```

(c) Les *programmes* sont des exemples autonomes, car ils sont utilisables tels quels, par simple copie. Certains programmes sont parfois présentés en plusieurs parties pour pouvoir introduire des explications intermédiaires. Dans ce cas, un commentaire `// ...` ou `/* ... */` montre comme d'habitude que du code a été omis. Voici un programme très court qui met en œuvre la fonction précédente.

Programme poly2.cpp

```
1
2  #include <iostream>
3  /*
4    Calcule la valeur d'un polynome du second degré,
5    de coefficients a,b,c au point x
6  */
7  double poly2(double x, double a, double b, double c)
8  {
9    double y ;
10   // Schéma d'HORNER
11   y = ((a * x) + b ) * x + c  ;
12   return y ;
13 }
14
15 int main()
16 {
17   double x,a,b,c ;
18   std::cout << "Entrez a b c puis x : " << std::endl  ;
19   std::cin >> a >> b >> c >> x ;
20   std::cout << "Valeur du polynome = " <<  poly2(x,a,b,c) << std::endl ;
21   return 0 ;
22 }
```

(d) Finalement, le texte affiché par un programme sera représenté dans ce livre avec une barre latérale de la façon suivante :

```
Valeur du polynome = 23.56
```

Tous les exemples de ce livre (et d'autres qui n'ont pas pu y figurer faute de place) peuvent être téléchargés sous la forme d'une archive depuis le site de l'éditeur à l'adresse suivante :

`https://www.editions-ellipses.fr/`

Chaque exemple est contenu dans un répertoire qui comporte les codes sources et les ressources nécessaires à leur mise en œuvre :

- Un script `bash` nommé `build.sh` pour construire les applications en ligne de commande sous Linux.
- Un fichier projet (extension `.pro`) utilisable par `creator`, l'excellent IDE venant avec la bibliothèque **Qt**. Cette bibliothèque est gratuite pour les usages non lucratifs. Utiliser `creator` est certainement la meilleure option, car cet IDE fonctionne aussi bien sous Linux et MacOS que sous Windows.

Toutes remarques et suggestions concernant ce livre peuvent être directement adressées à l'auteur par e-mail :

`progcpp23.ag@free.fr`

1.4 Pictogrammes utilisés dans le texte

Le texte de ce livre utilise plusieurs sortes de pictogrammes permettant d'attirer l'attention sur un point particulier.

 Introduit des explications concernant le code qui précède.

 Introduit un résumé qui rappelle les principaux points à assimiler.

 Introduit un renvoi vers la partie avancée du livre. Ceci permet au lecteur d'approfondir immédiatement le sujet en train d'être traité, s'il le désire.

 Introduit des explications techniques concernant l'implantation d'une fonctionnalité particulière. Lire ces explications permet d'avoir une connaissance plus profonde du langage.

 Signale une difficulté particulière, ou un piège à éviter.

 Signale une fonctionnalité du langage C qu'il est recommandé de ne pas utiliser en C++.

2 | À propos du langage C++

Le langage C++ est né des réflexions de Bjarne Stroustrup, qui a eu l'idée d'allier l'efficacité du langage C à la puissance d'expression permise par la programmation par objets. Le langage C a été créé dans les années 70 pour réaliser des logiciels système. Il s'agit d'un excellent langage, dépouillé mais performant, qui a eu et continue d'avoir un grand succès dans son domaine d'application. C'est pour cette raison que C++ a donc été conçu (au milieu des années 80) comme un surensemble du langage C, ce qui a permis de conserver le bénéfice de l'énorme logithèque de ce dernier.

Au fil des années, le langage a été doté de nombreuses fonctionnalités importantes : *l'héritage multiple*, la gestion des *exceptions* et surtout la *généricité*, qui a radicalement changé la manière de l'utiliser, et qui l'a rendu particulièrement extensible.

À cause de sa genèse et des similitudes syntaxiques, C++ est considéré par ceux qui ne le connaissent pas comme un « super C », mais c'est une erreur. Non seulement le C++ initial est très différent de C, mais le C++ moderne (c'est-à-dire à partir de C++11) permet une approche de la programmation très différente de celle que proposait son prédécesseur.

On dit parfois que C++ est un langage *multiparadigmes*, ce qui signifie qu'il supporte plusieurs façons d'aborder les problèmes et de créer les solutions correspondantes : il est à la fois *procédural* [1] (comme le C), *à objets* [2], et *générique* [3]. Ces différentes façons de voir les choses peuvent être combinées selon le type de problème à résoudre : on peut ainsi écrire des programmes dans un style *procédural pur*, ou qui seraient *à objets et génériques*, ou *procéduraux et génériques*, ou enfin combiner les trois paradigmes, ce qui est la possibilité la plus utilisée. Enfin, grâce à son approche de la généricité, C++ permet une certaine forme de *métaprogrammation* [4].

C++ est un langage évoluant en permanence, et son comité de normalisation est très actif. Les formes initiales du langage s'appellent C++98 et C++03. Les versions suivantes constituent le *C++ moderne* : il s'agit de C++11, C++14, C++17, C++20 et C++23 [5]. Cette dernière norme n'était d'ailleurs pas complètement supportée par tous les compilateurs lors de la réalisation de ce livre.

Malgré les différences, il y a un domaine dans lequel la philosophie du C++ est identique à celle du C : chaque fonctionnalité est conçue pour favoriser l'efficacité, et vous ne payez pas pour une fonctionnalité que vous n'utilisez pas. Pour cette raison, C++ est bien adapté aux projets complexes pour lesquels d'excellentes performances sont requises. C'est le cas par exemple des programmes et environnements graphiques, que nous utilisons tous les jours sur nos ordinateurs.

C++ est un langage très puissant dont les fonctionnalités de base sont abordables, mais il est tellement riche et vaste qu'il y a toujours des subtilités à y découvrir, même pour un programmeur expérimenté. C'est pourquoi, si à l'issue de la lecture de ce livre (ou de tout autre ouvrage traitant du même sujet), vous pensez avoir tout compris du C++, il est probable qu'on vous ait mal expliqué.

Bonne lecture !

1. La programmation procédurale est basée sur des assemblages de procédures (aussi appelées sous-programmes ou fonctions) qui implantent des algorithmes.
2. Ce sujet est traité au chapitre 18.
3. Ce sujet est traité au chapitre 27.
4. La métaprogrammation est une manière de programmer permettant dans certains cas d'obtenir des résultats à la compilation, donc sans exécuter de code. Ce sujet est traité au chapitre 56.
5. Le nom de chaque norme correspond à son année de publication. Il faut aussi savoir qu'il peut s'écouler plusieurs années entre la publication d'une norme et son support complet par les compilateurs. La prochaine version du langage s'appellera C++26.

Deuxième partie

Le socle de C++

3 | Composition des programmes C++

3.1 La notion de fonction en C++

Un programme rédigé en langage C++ comporte au moins une *fonction*. Une fonction est un groupe d'instructions qui réalisent un traitement à partir de données qu'on lui fournit. L'effet de ce traitement est généralement la production d'un résultat, comme pour une fonction mathématique. Par exemple, la fonction définie en mathématiques comme

$$f(x, y) \rightarrow x + y$$

reçoit deux nombres, en calcule la somme et retourne cette valeur. Il s'agit là du genre de fonction le plus classique, mais il en existe d'autres :

- Fonction sans paramètre : Il s'agit d'une fonction qui produit un résultat sans qu'on lui fournisse explicitement de données,

  $$heure() \rightarrow heureCourante$$

- Fonction sans résultat : Par extension du concept de fonction, le langage C++ permet aussi de définir des fonctions qui ne produisent pas de résultat. On dit dans ce cas que leur résultat est *vide* (ou `void`). Ceci pourrait être représenté comme :

  $$g(x) \rightarrow$$

  Une fonction à résultat vide peut être très utile en programmation. En effet, ses instructions, (comme d'ailleurs celles de toutes les autres fonctions) peuvent produire un *effet*[1] : elles peuvent par exemple lancer une fusée, éteindre une lampe, afficher un message ou plus simplement modifier l'état de la mémoire. La notion de fonction en programmation (et particulièrement en langage C++) est donc assez différente de ce qu'elle est en mathématiques.
- Fonction membre : Cette sorte de fonction est liée à la programmation par objet. Les fonctions membres (ou méthodes) existent donc en C++, mais pas en C. Leur spécificité est d'être attachées au traitement des objets. On n'en dira pas davantage dans cette partie du livre, car ce sujet sera traité à partir du chapitre 18 (page 189), qui introduit ce type de programmation.

3.1.1 Invocation des fonctions

Bien entendu, pour qu'une fonction soit utile, il faut qu'elle soit *appelée*[2], ce qui déclenche son exécution. C'est l'interaction entre les différentes fonctions qui composent le programme qui constitue l'activité de ce programme.
Le schéma de la figure 3.1 représente un programme très simple composé de 5 fonctions. Dans ce schéma, les flèches représentent des relations d'appel : la fonction `main` peut appeler les fonctions `b` et `d`. On notera que `d` n'appelle aucune autre fonction, mais que `b` peut appeler `a` qui peut appeler `c`, qui elle-même peut appeler `d`. On remarque donc que `d` peut être appelée de façon *directe* ou *indirecte*. Remarquez aussi que `a` est susceptible de s'appeler elle-même : il s'agit d'un *appel récursif*. Ce genre d'appel est traité au chapitre 16, page 159.

3.2 La fonction `main`

Comme la totalité du code constituant un programme se trouve dans des fonctions, il est nécessaire qu'une de ces fonctions soit lancée automatiquement et appelle les autres, directement ou indirectement. Il s'agit d'une fonction conventionnellement nommée `main`, et elle peut être la seule

1. On peut aussi utiliser le terme *effet de bord*.
2. On utilise aussi le terme *invocation* pour désigner un appel de fonction.

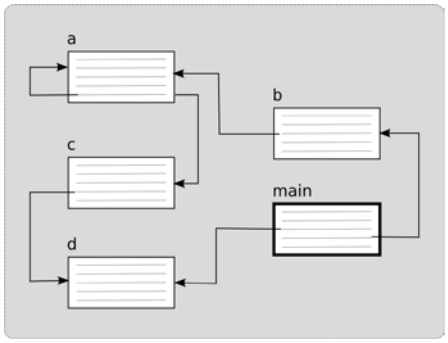


Figure 3.1 – Un programme simple composé de cinq fonctions.

du programme. Voici donc ce que vous devez retenir : si un programme écrit en langage C++ (ou C) ne comporte pas de fonction **main**, l'éditeur de liens [3] refusera de générer le fichier exécutable.

Toutes les fonctions **main** sont construites sur le modèle de celle qui suit. Pour l'instant, je ne donnerai pas d'autre explication sur les fonctions (qui seront traitées plus en détail au chapitre 6), car nous avons beaucoup d'autres choses à voir auparavant. La fonction **main** nous servira donc uniquement à accueillir les différents exemples, au fur et à mesure de leur présentation.

NeFaitRien/ne-fait-rien.cpp

```
1
2 #include <iostream>
3 // --------------- Exemple de fonction main vide (commentaire style C++)
4 int main()
5 {
6   /* Le code de la fonction doit se trouver ici (commentaire style C) */
7   return 0 ;
8 }
```

La fonction **main** débute ici à la ligne 4, et se termine à la ligne 8. Elle ne comporte aucune instruction, mais les lignes 3 et 6 comportent des *commentaires*. Ceux-ci n'ont aucun effet sur l'exécution des programmes, mais ils sont extrêmement importants, car ils permettent d'éclaircir ou de rappeler un point qui rendrait le programme difficile à comprendre. N'hésitez jamais à ajouter un commentaire, même si vous êtes la seule personne à lire votre programme.
Pour l'instant, ne vous préoccupez pas du reste du code : contentez-vous de recopier cet exemple tel quel avec votre éditeur de texte favori. Enregistrez ce code source dans le fichier **ne-fait-rien.cpp**, puis générez le programme exécutable. Par exemple, si vous travaillez en mode texte dans une console Linux, vous devez taper [4] :

```
g++ -o ne-fait-rien ne-fait-rien.cpp
```

Exécutez maintenant le programme en tapant :

```
./ne-fait-rien
```

S'il ne se passe rien, c'est plutôt bon signe. Bravo ! Vous venez d'exécuter votre premier programme en langage C++, celui qui ne fait rien.

Vous trouverez davantage des renseignements sur la fonction **main** au chapitre 10, page 113.

3. Si ce terme ne vous dit rien, lisez l'annexe A.1.2.3, page 799.
4. Les commandes de compilation/édition de liens sont décrites en annexe A.2, page 805.

4 | Les données et leur type

Ce qu'on appelle *donnée* dans le contexte des langages de programmation n'est autre que la « matière » sur laquelle travaillent les programmes. C'est pourquoi, avant de parler des outils mis à notre disposition par un langage de programmation, il faut préciser les concepts en relation avec les données.

4.1 Variables et constantes

Une donnée peut être une *constante* ou une *variable*. Ces deux termes rappellent de façon claire que la valeur d'une constante ne peut pas être modifiée, contrairement à celle d'une variable. Les variables sont désignées par un nom, tel que `hauteur` ou `xflr13`. Dans le vocabulaire informatique, on trouve souvent les termes *identificateur* ou *identifiant*, comme synonymes de nom. Les constantes n'ont pas de nom : elles peuvent être des valeurs intermédiaires résultant d'un calcul, ou des *littéraux* tels que `12.5` ou `"Bonjour"`.

4.2 À quoi sert une déclaration ?

La très grande majorité des langages compilés (tels C ou C++) exigent que les variables soient déclarées avant d'être utilisées. Une déclaration ressemble à une présentation : *« Compilateur, je te présente* `x`*, c'est une variable contenant une donnée de type entier »*. Grâce à cela, le compilateur pourra entreprendre les actions aboutissant à la création de la variable. Il saura aussi comment la manipuler lorsqu'il rencontrera l'identificateur `x`.

4.3 Qu'est-ce qu'un type ?

Les variables et constantes ont un type. Si vous n'êtes pas familier avec ce concept, lisez cette section. Sinon, vous pouvez passer directement aux sections suivantes qui décrivent les différents types disponibles en langage C++.

L'annexe A.1 (page 789) consacrée à la structure et au fonctionnement des ordinateurs montre que toutes les informations stockées en mémoire sont représentées de la même manière [1]. Si l'on s'intéresse à un octet particulier pris au hasard, il est absolument impossible de dire ce qu'il représente, car il ne comporte aucune information pouvant nous renseigner.

Or, selon que cet octet contribue à représenter une adresse, un entier, un réel ou autre chose, le programme devra le manipuler de différentes façons. Par exemple, la façon de multiplier deux réels est différente de celle qui permet de multiplier deux entiers.

L'information décrivant la nature d'une entité informatique s'appelle un *type*. Connaître le type d'une entité peut être utile au moment de l'exécution du programme, mais pour les langages compilés, c'est surtout lors de la compilation que le type est exploité. Il permet au compilateur de :

(a) faire en sorte que la place nécessaire et suffisante pour stocker chaque entité soit réservée. Corollairement, ceci lui permet d'associer à chaque identificateur utilisé (`x`, `y`, `masse`, `vitesse`, `prix`, etc.) une *adresse* en mémoire.

(b) fabriquer le code machine manipulant correctement cette entité. C'est tout de même le point le plus important !

1. Avec les technologies actuelles, il s'agit d'une représentation binaire.

(c) vérifier que le programmeur utilise bien les données de façon cohérente. Par exemple, l'addition de deux nombres réels semble cohérente. Par contre, additionner un nombre réel avec un texte est probablement une erreur.


Le dernier point cité est très important : la notion de type a pour effet *d'introduire du sens* dans vos programmes, ce qui permet au compilateur d'en vérifier (dans une certaine mesure) la cohérence. Dans le jargon informatique, on utilise le terme *sémantique* pour désigner le sens attaché à un type.

La sémantique des types (quels qu'ils soient) peut être définie par :

(a) le langage (c'est le cas des types primitifs disponibles en C ou C++),

(b) la bibliothèque standard du langage (en C++ uniquement),

(c) l'utilisateur (également en C++).


La sémantique des types permet de détecter automatiquement beaucoup d'erreurs de programmation, mais ne vous faites pas d'illusions : il existe tant de façons de se tromper qu'il reste très facile d'écrire des programmes faux. Par conséquent, le fait qu'un programme ait été compilé sans message d'erreur n'est absolument pas une garantie que ce programme soit correct, malgré l'apport du typage.

4.4 Les types primitifs

On dira qu'une entité est d'un type *fondamental* (ou *primitif simple*) si son type ne s'appuie pas sur l'existence d'un autre type. Il existe en C++ trois sortes de *types primitifs simples* : les *entiers*, les *réels* et les *booléens*. Il existe aussi des types *primitifs composés*, tels que les *pointeurs*, les *tableaux* et les *structures*. Le premier sera traité ici, mais les autres seront traités au chapitre 8, page 83.

Qu'elles soient d'un type simple ou composé, les entités primitives ont en commun de pouvoir être manipulées par des opérateurs prédéfinis du langage.


En C++, déclarer une variable d'un type *primitif* ne l'initialise pas, à moins que le programmeur le demande explicitement. Vous ne devez donc jamais supposer qu'une telle variable possède *par défaut* une valeur particulière [a]. En revanche, on verra qu'il est possible de garantir que tous les *objets* (que vous pouvez considérer comme des variables d'un type non primitif) soient initialisés.

a. Ceci est un héritage contestable du langage C.

4.4.1 Entiers

Les nombres entiers utilisés dans les ordinateurs ne représentent qu'approximativement l'ensemble mathématique des entiers, car on ne sait représenter qu'un nombre fini d'entiers. Le langage C++ utilise généralement la représentation de la machine cible. Comme presque tous les ordinateurs utilisent la même représentation, cela limite les surprises. Les entiers peuvent être signés (entiers relatifs) ou non signés (entiers naturels). Les entiers signés sont presque toujours représentés par un procédé nommé *complément à deux*. À partir de C++20 , ce procédé est même obligatoire.

Le langage ne propose pas moins de cinq types entiers qui s'appellent `char`, `short`, `int`, `long` et `long long`. Sans spécification explicite, ils sont tous signés, à l'exception de `char`, qui peut être signé ou non selon le compilateur. Remarquez toutefois que la plupart des compilateurs considèrent que les `char` sont signés par défaut.

Le tableau de la figure 4.1 indique les intervalles de valeurs représentables par des entiers codés sur N bits.

Type d'entier	Plus petite valeur	Plus grande valeur
Entiers non signés	0	$(2^N) - 1$
Entiers signés	$-(2^{N-1})$	$(2^{N-1}) - 1$

Figure 4.1 – Intervalle des valeurs représentables par des entiers codés sur N bits

Pour obtenir la version non signée de ces types, il suffit de préfixer leur nom par le mot-clé **`unsigned`**, comme dans **`unsigned long`**. La taille de l'entier résultant est la même que celle de l'entier signé correspondant. Pour mettre en évidence le fait qu'un entier est signé, vous pouvez aussi utiliser le préfixe **`signed`**, mais à part pour les **`char`**, ceci est redondant. Le langage impose quelques contraintes concernant la quantité de mémoire utilisée pour représenter les entiers : les **`char`** doivent être par définition codés sur un octet, et la taille minimale des **`int`** est de 2 octets. À part cela, les relations suivantes entre les différentes tailles doivent être vérifiées :

$$taille(\texttt{char}) \leq taille(\texttt{short}) \leq taille(\texttt{int}) \leq taille(\texttt{long}) \leq taille(\texttt{long long})$$

Pour fixer les idées, le tableau de la figure 4.2 indique les tailles typiquement utilisées, dans deux environnements classiques. Ces informations sont purement indicatives : il est bon de consulter les spécifications de votre compilateur avant d'utiliser un type plutôt qu'un autre. Cependant, il est probable que ces informations correspondent exactement à votre système.

Type C++ de l'entier	Ordinateur personnel	Microcontrôleur
`char`	1	1
`short` (ou `short int`)	2	2
`int`	4	2
`long` (ou `long int`)	4	4
`long long`	8	4

Figure 4.2 – Tailles typiques (en octets) des entiers supportés par les compilateurs C++, dans deux environnements très différents.



Avant de passer à un exemple, voici quelques précisions importantes concernant les types **`char`** et **`int`**, qui sont très utilisés.

- Le nom **`char`** est l'abréviation de « *character* ». Le choix de ce nom est sans doute une décision de conception malheureuse, car il fait invariablement croire aux débutants que ce type de donnée sert forcément à représenter des caractères. En fait, les **`char`** peuvent effectivement servir à cela, mais aussi à tout autre chose. Le **`char`** est simplement un entier, sur lequel vous pouvez faire exactement les mêmes opérations que sur les autres entiers. Sa seule spécificité est d'être toujours codé sur 8 bits [2], ce qui ne permet de représenter que 256 valeurs.
- Le type **`int`** est très souvent équivalent à **`short`** ou **`long`**, selon la machine. On peut donc se demander à quoi il sert, puisqu'il semble redondant. Ce type a été créé pour supporter de façon efficace les calculs en nombres entiers sur la machine cible. Pour cela, les variables **`int`** ont, en principe, exactement la même taille que les registres de données du processeur, ce qui assure théoriquement un fonctionnement optimal sur celui-ci. Ceci explique sa fréquente utilisation.

2. En théorie, les **`char`** pourraient cependant être codés sur plus de 8 bits, mais les architectures de processeurs exploitant cette possibilité sont extrêmement rares.

4.4.1.1 Déclaration des variables entières

Déclarer une variable de type entier est très simple. Cette déclaration sera de la forme :

type_de_la_variable nom_de_la_variable;

Par exemple :

```
int hauteur ;    /* hauteur est un int */
short vitesse ;  /* vitesse est un short */
```

4.4.1.2 Exemple d'utilisation de nombres entiers

Calculs entre nombres entiers

```
1
2  #include <iostream>
3
4  int main()
5  {
6    int a ;   // a est un int
7    int b,c ; // b et c sont aussi des int
8
9    b = 2 ;   // mettons la valeur 2 dans b
10   c = 10 ;  // et la valeur 10 dans c
11   a = b * c ;  // mettons le produit de b par c dans a
12
13   std::cout << a << std::endl ; // affichons la valeur de a
14
15   return 0 ;
16 }
```



Cet exemple est très court, mais comme c'est le premier vrai programme de ce livre il y a beaucoup à en dire. On va donc le décomposer pour l'expliquer.

```
  int a ;   // a est un int
  int b,c ; // b et c sont aussi des int
```

Ces deux lignes *définissent* les variables `a`,`b` et `c`. Elles demandent au compilateur de faire ce qu'il faut pour qu'un petit morceau de mémoire soit réservé pour chacune de ces variables. Elles précisent aussi le type de chaque variable, ce qui permet au compilateur de générer un code approprié à ce type.

D'un point de vue syntaxique, chaque instruction est terminée par un « ; », et vous avez le droit de mettre plusieurs instructions sur la même ligne, ce qui permet de mettre en évidence le fait qu'elles sont logiquement liées. Hors de cette situation, n'abusez pas des lignes de code longues, car elles rendent les programmes difficiles à lire.

```
  b = 2 ;      // mettons la valeur 2 dans b
  c = 10 ;     // et la valeur 10 dans c
  a = b * c ;  // mettons le produit de b par c dans a
```

Ces instructions réalisent des *affectations*. L'opérateur `=` sert ici à transférer la valeur qui est à sa droite dans la variable qui est à sa gauche. L'opérateur `=` sert donc à changer le contenu de cases en mémoire. L'opérateur `*` sert quant à lui à faire le produit des valeurs des variables qui l'entourent, qu'on appelle des *opérandes*. Finalement, vous l'avez compris, la variable `a` devrait contenir la valeur 20 à l'issue de l'exécution de ce morceau de code.

```
std::cout << a << std::endl ; // affichons la valeur de a
```

Cette instruction (qui est composée d'une seule expression) permet d'afficher sur l'écran la valeur de `a`, suivie d'un saut de ligne. Dans cette expression, l'opérateur `<<` permet d'écrire son opérande de droite sur le flux représenté par son opérande de gauche.
Comme l'*objet*[a] prédéfini `std::cout` représente l'écran, l'expression affiche `a` sur l'écran. La même opération est ensuite effectuée avec `std::endl`, qui représente un saut de ligne.

a. Pour l'instant, vous pouvez considérer les objets comme des variables ayant des propriétés un peu particulières.



Vous trouverez au chapitre 7 (page 79) des explications plus détaillées sur le fonctionnement des entrées-sorties sur flux utilisant les opérateurs `<<` et `>>` .

Exécutons maintenant ce programme grâce aux deux commandes qui suivent. La première permet de générer le fichier exécutable, alors que la seconde permet de lancer le programme. Le nom du fichier source est ici `calculs-entiers.cpp` .

```
g++ -o calculs-entiers calculs-entiers.cpp
./calculs-entiers
```

```
20
```

4.4.2 Entiers techniques

Il existe en C++ quelques types entiers qui ne sont pas dédiés au calcul numérique, mais à la gestion des structures de données. I s'agit de `size_t`, `ssize_t` et `ptrdiff_t`. Faute d'une terminologie officielle, on les appellera ici *entiers techniques*. Leur usage n'est pas absolument indispensable, mais recommandé si l'on veut éviter d'éventuels débordements. En outre, il faut savoir qu'ils sont utilisés par certains opérateurs (l'opérateur `sizeof` retourne par exemple un entier de type `size_t`), ou par des fonctions de la bibliothèque standard.

(a) `size_t` est un type non signé permettant de représenter la taille de n'importe quelle donnée. Il est aussi utilisable pour déclarer les indices permettant d'accéder aux éléments des conteneurs n'acceptant pas les indices négatifs. Le conteneur `vector` (section 32.1, page 397) fournit par exemple pour cet usage un type synonyme de `size_t`.

(b) `ssize_t` est un type signé jouant le même rôle que `size_t` pour les conteneurs manipulables via des indices positifs ou négatifs. Cela peut être le cas lorsqu'on manipule des tableaux au travers d'un pointeur pour obtenir des indices décalés (voir section 14.4, page 146).

(c) `ptrdiff_t` est un type signé permettant de représenter la différence de deux pointeurs. Reportez vous à la section 14.1 (page 141) consacrée à l'arithmétique des pointeurs pour en savoir davantage sur ce sujet.

Depuis C++23 , il est possible d'utiliser les littéraux de types `size_t` ou `ssize_t`. Ceci permet de supporter la déduction automatique de type, lorsqu'une donnée est déclarée avec `auto` (cette fonctionnalité est décrite en section 17.5, page 181). La table de la figure 4.3 indique les suffixes à utiliser.

Type désiré	Suffixe	Exemple
`ptrdiff_t`	*inexistant*	
`size_t`	*uz* ou *UZ*	`0uz`, `4UZ`
`ssize_t`	`z` ou `Z`	`0z`, `0xFFFFFFFFZ`

Figure 4.3 – Les différents suffixes permettant de préciser le type d'une constante entière technique.

4.4.3 Booléens

Le type `bool` permet en C++ de représenter les valeurs logiques. Les variables de ce type ne peuvent prendre que deux valeurs : `true` (*vrai*), ou `false` (*faux*). Tous les opérateurs de comparaison retournent par exemple un résultat de type `bool`.

Ce type n'existait pas dans les premières versions de C++, qui utilisaient la convention du langage C pour la représentation des valeurs logiques. En C, les valeurs logiques étaient représentées par un `int`, avec la convention suivante : `0` signifie *faux*, et toutes les autres valeurs signifient *vrai*. Par exemple, les opérateurs de comparaison retournaient `0` (*faux*) ou `1` (*vrai*). On en parle ici, parce que le type `bool` est compatible avec l'ancienne représentation, qui est toujours utilisable. Par exemple :

```
bool a, b, c, d ;
a = true ;  // a vaut true
b = 1 ;     // b vaut true
c = 34 ;    // c vaut aussi true
d = false ; // d vaut false
```

Si vous affichez les valeurs des variables précédentes, vous aurez d'ailleurs une surprise, car l'affichage des booléens utilise par défaut la représentation numérique.

```
std::cout <<  a << " " << b << " " << c << " " << d << std::endl ;
```

```
1 1 1 0
```

Toutefois, il est facile de configurer le mode d'affichage[3], de manière à ce que les booléens soient imprimés d'une manière spécifique :

```
std::cout << std::boolalpha ; // configure le mode d'affichage des bool
std::cout <<  a << " " << b << " " << c << " " << d << std::endl ;
```

```
true true true false
```



Pour des raisons de rétrocompatibilité, il est possible d'utiliser une expression de type `int` là ou une expression logique est attendue : celle-ci est automatiquement convertie vers le type `bool`. Et ceci est également valable pour les pointeurs, un pointeur nul étant considéré comme *faux*. Voici un exemple où un entier est utilisé comme une expression logique :

```
int k = f() ;  // f est une fonction à résultat entier
if(k)          // contexte d'un test : k est automatiquement converti en bool
{
  // ...
}
```

4.4.4 Réels

Les nombres réels manipulés dans les ordinateurs ne sont qu'une approximation des nombres réels du monde mathématique. Ils sont représentés par un procédé appelé *virgule flottante*, que je ne détaillerai pas ici. Pour cette raison, ces nombres s'appellent *nombres à virgule flottante*, ou plus simplement *flottants*. Comme pour les entiers, le résultat de la représentation est imparfait : on ne sait représenter qu'un nombre limité de réels. Aussi bizarre que cela paraisse, un flottant permet par exemple de représenter exactement le nombre $\frac{1}{4}$, mais pas le nombre $\frac{1}{3}$.
Le schéma de la figure 4.4 montre l'axe des nombres réels, sur lequel les zones grisées contiennent les nombres représentables. Le nombre 0 est représentable, mais les autres nombres situés dans l'intervalle $[-min \cdots + min]$ ne le sont pas, tout comme ceux dont la valeur absolue est $> max$. Il faut noter que les réels se trouvant dans les zones grisées ne sont pas tous représentables, puisqu'il y en a une infinité !

3. Les manipulateurs d'entrées/sorties, tels que `boolalpha` sont décrits en section 29.11, page 360

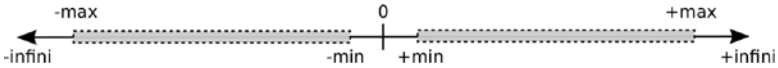


Figure 4.4 – Zones de l'axe des réels dans lesquelles les nombres flottants sont représentables.

Les limites de ces différentes zones dépendent du type de nombre flottant utilisé. Trois variantes de flottants existent en langage C et C++ : les `float`, les `double` et les `long double`. Leur représentation est dépendante du compilateur, mais il est fréquent qu'ils soient codés sur 4, 8 et 16 octets, selon la norme IEEE-754 créée par l'*Institute of Electrical and Electronics Engineers*.

À partir de C++23 , vous pouvez utiliser des types flottants dont la taille et la représentation sont imposées par la norme [4] : il s'agit de `float16_t`, `float32_t`, `float64_t`, `float128_t` et `bfloat16_t`. Ces nouveaux types utilisent respectivement *toujours* 2, 4, 8, 16 et 2 octets de mémoire [5].

Le tableau de la figure 4.5 montre les limites des intervalles de valeurs représentables.

Type C++	taille (octets)	min	max
`float` (ou `float32_t`)	4	$\approx 1.2 \times 10^{-38}$	$\approx 3.4 \times 10^{38}$
`double` (ou `float64_t`)	8	$\approx 2.2 \times 10^{-308}$	$\approx 1.7 \times 10^{308}$
`long double` (ou `float128_t`)	16	$\approx 3.4 \times 10^{-4932}$	$\approx 1.2 \times 10^{4932}$

Figure 4.5 – Exemples d'intervalles de valeurs absolues représentables par des nombres flottants, selon la norme IEEE-754. Ces valeurs sont arrondies.

La précision des nombres représentables est limitée. Par exemple, le type `float` permet de coder des nombres avec 6 digits décimaux significatifs, ce qui est souvent insuffisant. Nous disposons de 15 digits significatifs avec un `double`, et de 18 avec un `long double`. Comme le type `float` est assez peu précis, il est rarement utilisé : c'est pour cela que le type flottant standard des langages C et C++ est le `double`.

4.4.4.1 Déclaration des variables réelles

La déclaration d'une variable réelle est analogue à celle d'un entier.

type_de_la_variable nom_de_la_variable;

Par exemple :

```
double x ;  /* x est un double */
float k ;   /* k est un float */
```

4.4.4.2 Exemple d'utilisation de réels

Calculs utilisant des nombres flottants

```
1
2 #include <iostream>
3
4 int main()
```

4. Utiliser ces types permet de garantir que vos données resteront représentables lors du portage des programmes.
5. Le type `bfloat16_t` est une version de `float16_t` offrant moins de précision et davantage d'amplitude

```
{
  int a,b ;
  double x,y,z ;
  a = 10 ; b = 3 ;
  x = 10 ; y = 3 ;
  z = a / b ;
  std::cout << z << std::endl ;
  z = x / y ;
  std::cout << z << std::endl ;

  return 0 ;
}
```

Le résultat obtenu sur ma machine est le suivant :

```
3
3.33333
```

Cet exemple ressemble au précédent, mais réalise des divisions. Cette opération est appliquée de la même manière à des `int` et des `double`. Dans les deux cas, le résultat est affiché grâce à l'opérateur `<<` , comme dans l'exemple concernant les nombres entiers.

Ce code a été conçu pour donner un résultat surprenant. En effet, la valeur du quotient `a/b` est `3`, et celle du quotient `x/y` est `3.333333`. Le premier résultat vient du fait qu'en C++, un calcul entre entiers donne *toujours* un entier. Le calcul fournit donc la partie entière du quotient, qui est ensuite stockée dans une variable de type `double`.

4.4.5 Types des constantes

Les exemples précédents utilisent des *variables* (`x`, `y`, etc.) et des *constantes littérales* (`3`, `10`, etc.). Le type de chaque variable est obligatoirement fourni à sa déclaration, ce qui permet au compilateur de la manipuler correctement. Mais comment le type des constantes littérales est-il spécifié ? La réponse est simple : la façon de les écrire détermine non seulement leur valeur, mais aussi leur type. Il existe deux familles de constantes numériques : les *entiers* et les *flottants*. Toute constante numérique qui comporte un point ou un exposant décimal est de type flottant. Sinon elle est de type entier, comme dans les exemples ci-dessous.

```
i = 3 ;     /* 3 est un entier (int) */
a = 4.5 ;   /* 4.5 est un flottant (double) */
b = 5E3 ;   /* 5E3 est un flottant (5000.0, double)  */
c = 5e3 ;   /* 5e3 est un flottant (5000.0, double)  */
d = 4.5E3 ; /* 4.5E3 est un flottant (4500.0, double)  */
```

4.4.5.1 Écriture des constantes entières sous forme numérique

Les littéraux entiers peuvent être écrits en base 10, en base 2, en base 8 ou en base 16. Cette base est spécifiée en utilisant un préfixe. Le tableau de la figure 4.6 indique les différents préfixes disponibles, et les écritures correspondantes.

Remarquez que ces préfixes commencent par la lettre `0` (zéro) et non `O` (o majuscule).

En pratique, les bases d'écriture décimales et hexadécimales sont très utiles. L'écriture en base 2 peut aussi être utile dans certains domaines, tels que celui des logiciels qui accèdent directement au matériel. Par contre, l'écriture en base 8 est peu utile, et elle est une source classique de confusion. Par exemple `010` signifie 10_8, c'est-à-dire 8_{10} et non 10_{10} comme on pourrait le croire [6].

6. La notation N_b signifie N en base b.

Base d'écriture	Préfixe	Exemples
10 (*décimal*)	*sans préfixe*	`-34`, `0`, `4`
2 (*binaire*) à partir de C11	`0b`	`0b111`, `0b10110`
8 (*octal*)	`0`	`07`, `010`, `0377`
16 (*hexadécimal*)	`0x`	`0x100`, `0x1C`, `0x1BAFFE`

Figure 4.6 – Expression d'une constante entière dans quatre bases de numération.

N'utilisez donc l'écriture octale que si vous avez de sérieuses raisons de le faire.

Voici différentes façons d'écrire la constante `255`, en l'exprimant dans différentes bases de numération.

```
i = 255 ;         /* 255, decimal */
j = 0377 ;        /* aussi 255, octal */
k = 0xFF ;        /* encore 255, hexadécimal */
l = 0b11111111 ;  /* toujours 255, binaire */
```


La base utilisée pour écrire une constante n'a aucune incidence sur la représentation en machine de cette constante : compte tenu de la technologie des ordinateurs actuels, elle est presque toujours représentée en base 2. Les contenus des quatre variables de l'exemple précédent sont donc physiquement identiques, pour une machine donnée.

4.4.5.2 Écriture des entiers sous forme « caractère »

Les constantes entières peuvent également être écrites sous forme non numérique : par exemple `'A'` est un entier de type `char` dont la valeur est celle du code [7] de la lettre `A`. Ce genre de constante doit être vu comme un moyen portable d'obtenir le code d'un caractère particulier.


En langage C, le type des constantes écrites de cette manière est `int`, et non `char`.

Les codes des caractères non imprimables [8] peuvent être obtenus grâce à une séquence d'échappement prédéfinie commençant par `\`. Par exemple, la séquence `'\n'` correspond au caractère « saut de ligne » (*newline* en anglais). Le tableau de la figure 4.7 donne l'écriture des caractères non imprimables les plus fréquents.

Le petit morceau de programme suivant permet d'afficher les valeurs de quelques constantes écrites sous forme caractère. L'utilisation de l'opérateur `sizeof` [9], qui permet d'obtenir la taille (en octets) du résultat d'une expression permet de montrer que le type de la constante `'x'` est bien `char`, qui occupe 1 octet sur ma machine.

```
int code  = 'A' ;
std::cout << code << std::endl ;
code = '\n' ;
std::cout << code << std::endl ;
/* Quelle est la taille du résultat de l'expression 'x' ? */
std::cout << sizeof 'x' << std::endl ;
```

Ce qui affiche sur ma machine :

7. Le code d'une lettre est simplement un entier qui représente conventionnellement cette lettre. Sur les machines utilisant le code ASCII ou UNICODE (c'est-à-dire presque toutes), la valeur correspondant à la lettre `A` est 65

8. Un caractère non imprimable est un caractère auquel ne correspond aucun dessin (ou glyphe).

9. Bien que `sizeof` ne ressemble pas aux opérateurs auxquels nous sommes habitués, il en est réellement un, au même titre que `+`, `-`, etc.