

Alain Gibaud

Programmer en C++

Des premiers pas à la maîtrise de C++23

2^e édition



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 <code>main</code>	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	<code>const</code> et les pointeurs	52
4.5.5	Quand utiliser <code>const</code> ?	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 <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code>	55
5.1.2	Opérateur <code>%</code>	55
5.1.3	Opérateurs <code>++</code> et <code>--</code>	56
5.2	Opérateurs sur bits	57
5.2.1	Opérateur <code>~</code>	57
5.2.2	Opérateur <code>&</code>	57
5.2.3	Opérateur <code> </code>	58
5.2.4	Opérateur <code>^</code>	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 séquençement (,)	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 <code>iostream</code>	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 <code>if .. else</code>	97
9.2.2 Tests sans clause <code>else</code>	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 (<code>while</code>)	101
9.3.2 Boucle avec test en fin (<code>do .. while</code>)	102
9.3.3 Boucle <code>for</code>	103
9.3.4 Boucle <code>for</code> contre boucle <code>while</code>	105
9.4 Sélection (<code>switch/case/default</code>)	106
9.4.1 Rôle de l'instruction <code>break</code> dans l'instruction <code>switch</code>	107
9.4.2 Exemples d'utilisation de <code>switch</code>	107
9.4.3 Utiliser <code>switch</code> ou <code>if</code> ?	108
9.5 Autres instructions contrôlant le flux d'exécution	110
9.5.1 Instruction <code>break</code>	110
9.5.2 Instruction <code>continue</code>	110
9.5.3 <code>break</code> , <code>continue</code> et programmation structurée	111
9.5.4 Instruction <code>goto</code>	112
10 La fonction <code>main</code>	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 <code>main</code>	115
10.4 La fonction standard <code>exit</code>	115

11 Le préprocesseur	117
11.1 Qu'est-ce que le préprocesseur?	117
11.2 Inclure un fichier avec <code>#include</code>	118
11.3 Définir une macro avec <code>#define</code>	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 <code>#undef</code>	123
11.5 Compilation conditionnelle	123
11.5.1 Directives <code>#if</code> , <code>#else</code> , <code>#elif</code> et <code>#endif</code>	124
11.5.2 Directives <code>#ifndef</code> , <code>#ifdef</code> , <code>#elifdef</code> et <code>#elifndef</code>	124
11.6 Autres <code>#</code> directives	125
11.6.1 Affichage d'un message d'erreur fatale avec <code>#error</code>	125
11.6.2 Affichage d'un message d'avertissement avec <code>#warning</code>	125
11.6.3 Transformation d'un paramètre de macro en chaîne avec <code>#</code>	126
11.6.4 Concaténation de paramètres de macro avec <code>##</code>	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 <code>typedef</code> ou <code>using</code>	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 <code>new</code>	153
15.3 Opérateur <code>new[]</code>	154
15.4 Opérateur <code>delete</code>	154
15.5 Opérateur <code>delete[]</code>	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 Implémentations récursives efficaces, mais peu intéressantes	162
16.3.2 Implémentations récursives peu efficaces	162
16.3.3 Implémentations récursives intéressantes	163
16.4 Fonctions à la fois <code>inline</code> 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 (<code>namespace</code>)	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 <code>std</code>	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 <code>auto</code>	181
17.6 Déclaration par emprunt de type avec <code>decltype</code>	181
17.7 Les transtypages à partir de C++11	181
17.8 Le qualificatif <code>constexpr</code>	182
17.9 Le qualificatif <code>constexpr</code>	184
17.10 Test statique avec <code>if constexpr</code>	185
17.11 Vérifications statiques avec <code>static_assert</code>	185
17.12 Boucle <code>for</code> 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 <code>this</code>	205
20.6 Méthodes statiques	207
20.7 Méthodes constantes	208
20.8 Fonctions, méthodes et classes amies (<code>friend</code>)	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 <code>inline</code>	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 : <code>()</code> ou <code>{}</code> ?	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 <code>mutable</code>	234
22.8 <code>const_cast</code> contre <code>mutable</code>	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 <code>=</code> et <code>&</code> 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 <code>operator+</code> (deux opérandes)	240
23.4.2 Méthode <code>operator-</code> (un seul opérande)	241
23.5 Exemple de surcharge d'opérateurs par fonctions globales	241
23.5.1 Fonction globale <code>operator+</code> (deux opérandes)	241
23.5.2 Fonction globale <code>operator-</code> (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 <code>=</code>	245
23.10.2 Opérateur de sortie (<code><<</code>)	250
23.10.3 Opérateur d'entrée (<code>>></code>)	252
23.10.4 Opérateurs d'entrée et de sortie : les bonnes pratiques	254
23.11 Autres opérateurs	255
23.11.1 Opérateur <code>[]</code>	255
23.11.2 Opérateur <code>[]</code> en C++23	256
23.11.3 Opérateur <code>*</code>	257
23.11.4 Opérateur <code>()</code> 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 <code>explicit</code>	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 <code>Noeud</code>	293
26.7.3 Les classes <code>Noeud1</code> et <code>Noeud2</code>	294
26.7.4 La classe <code>Constante</code>	295
26.7.5 Classes <code>Somme</code> , <code>Produit</code> et <code>Negation</code>	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 <code>override</code>	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 <code>PoLynome</code> générique	316
27.8.2 Implantation de la classe <code>PoLynome</code> 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 <code>try</code>	330
28.2.2 Instruction <code>throw</code>	330
28.2.3 Instruction <code>catch</code>	331
28.2.4 Exemple	331
28.2.5 Exceptions standards	333
28.2.6 Mode de passage de l'exception à un bloc <code>catch</code>	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 <code>ios_base</code>	343
29.1.2 <code>ios</code>	343
29.1.3 <code>ostream</code>	343
29.1.4 <code>istream</code>	343
29.1.5 <code>iostream</code>	344
29.1.6 <code>ostringstream</code>	344
29.1.7 <code>ofstream</code>	344
29.1.8 <code>istringstream</code>	344
29.1.9 <code>ifstream</code>	344
29.1.10 <code>stringstream</code>	344
29.1.11 <code>fstream</code>	344
29.1.12 Flux standards préouverts	344
29.2 Les différents états des flux	345
29.2.1 <code>fail</code> , <code>bad</code> , <code>eof</code> et <code>good</code>	345
29.2.2 <code>operator void *</code> et <code>operator bool</code>	347
29.2.3 Modifier l'état d'un flux avec <code>clear</code>	347
29.3 Les différents modes d'ouverture des flux	349
29.4 <code>istream</code>	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 <code>ifstream</code>	354
29.5.1 Construction et destruction	354
29.5.2 <code>rdbuf</code>	354
29.5.3 <code>open</code>	354
29.5.4 <code>close</code>	354
29.5.5 <code>is_open</code>	354
29.5.6 Exemple	355
29.6 <code>istringstream</code>	355
29.6.1 Construction et destruction	355
29.6.2 <code>rdbuf</code>	355
29.6.3 <code>str</code>	356
29.6.4 Exemple	356
29.7 <code>ostream</code>	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 <code>ofstream</code>	358
29.8.1 Construction et destruction	358
29.8.2 <code>rdbuf</code>	358
29.8.3 <code>open</code>	358
29.8.4 <code>close</code>	358
29.8.5 <code>is_open</code>	358
29.9 <code>ostringstream</code>	358
29.9.1 Construction et destruction	358
29.9.2 <code>rdbuf</code>	358
29.9.3 <code>str</code>	359

29.10	<code>iostream</code> , <code>fstream</code> et <code>stringstream</code>	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 <code><ios></code>	360
29.11.4	Manipulateurs déclarés dans <code><istream></code>	362
29.11.5	Manipulateurs déclarés dans <code><ostream></code>	363
29.11.6	Manipulateurs déclarés dans <code><iomanip></code>	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 <code>format</code>	368
29.13.1	Fonction <code>format</code>	369
29.13.2	Fonction <code>format_to</code>	374
29.13.3	Fonction <code>print</code>	374
29.14	Extension de la bibliothèque <code>format</code>	375
29.14.1	Prérequis	375
29.14.2	API d'extension de la bibliothèque <code>format</code>	375
29.14.3	Formatage du type <code>Monome</code>	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 <code>vector</code>	397
32.1.1	Description générale	397
32.1.2	Modèle d'extension	397
32.1.3	Construction d'un <code>vector</code>	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 <code>vector</code>	400
32.1.8	Exemple	400
32.2	Le conteneur <code>deque</code>	404
32.2.1	Description générale	404
32.2.2	Modèle d'extension	404
32.2.3	Construction d'un <code>deque</code>	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 <code>array</code>	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 <code>string</code> (<code>basic_string<char></code>)	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 <code>string</code>	414
32.4.8	Exemple	415
32.5	Les conteneurs <code>forward_list</code> et <code>list</code>	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 <code>map</code> et <code>multimap</code>	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 <code>unordered_map</code> et <code>unordered_multimap</code>	431
32.7.1	Description générale	432
32.7.2	Différence entre <code>unordered_map</code> et <code>unordered_multimap</code>	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 <code>flat_map</code> et <code>flat_multimap</code> (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	<code>inplace_merge</code>	483
33.6.3	<code>includes</code>	483
33.6.4	<code>set_difference</code>	483
33.6.5	<code>set_symmetric_difference</code>	483
33.6.6	<code>set_intersection</code>	484
33.6.7	<code>set_union</code>	484
33.6.8	Exemples	484
33.7	Algorithmes sur les tas (<i>heaps</i>)	486
33.7.1	Définition	486
33.7.2	<code>make_heap</code>	487
33.7.3	<code>is_heap</code>	487
33.7.4	<code>is_heap_until</code>	487
33.7.5	<code>push_heap</code>	487
33.7.6	<code>pop_heap</code>	487
33.7.7	<code>sort_heap</code>	487
33.7.8	Exemples	487
34	Type <code>initializer_list</code>	489
34.1	Création d'un <code>initializer_list</code>	489
34.2	Interface du modèle <code>initializer_list</code>	489
34.3	Exemple	490
35	Liaison structurée (<i>Structured binding</i>)	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 <code>{}</code> ou <code>()</code> ?	511

39 Allocation dynamique	513
39.1 Personnaliser <code>new</code> , <code>new[]</code> , <code>delete</code> et <code>delete[]</code>	513
39.1.1 Opérateur <code>new</code>	513
39.1.2 Opérateur <code>delete</code>	514
39.1.3 Opérateur <code>new[]</code>	514
39.1.4 Opérateur <code>delete[]</code>	515
39.1.5 Exemple	515
39.2 Opérateur <code>new</code> avec placement	517
40 Opérateurs	521
40.1 Opérateur <code>=</code>	521
40.1.1 Implantation de l'opérateur <code>=</code> par l'idiome <code>swap</code>	521
40.1.2 Gestion des auto-affectations	523
40.1.3 L'opérateur <code>=</code> retourne une lvalue en C++	523
40.2 Opérateurs <code>++</code> et <code>--</code>	523
40.2.1 Que retournent <code>++</code> et <code>--</code> ?	523
40.2.2 Distinction entre les formes préfixées et postfixées	524
40.2.3 Exemple de surcharge de l'opérateur <code>++</code>	525
40.2.4 Un opérateur <code>++</code> postfixé gratuit?	527
40.3 Opérateur <code>-></code>	528
40.3.1 Fonctionnement de l'opérateur <code>-></code>	528
40.3.2 Un mandataire basé sur l'opérateur <code>-></code>	528
40.3.3 Un pointeur intelligent basé sur l'opérateur <code>-></code>	530
40.4 Opérateur <code>""</code>	532
40.4.1 Principes de mise en œuvre	533
40.4.2 Exemple : littéral de type <code>short</code>	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 <code>constexpr</code>	536
40.5 Opérateurs statiques (C++23)	537
40.6 Opérateur de comparaison à trois voies <code><=></code> (C++20)	538
40.6.1 Introduction	538
40.6.2 Opérateur <code><=></code> par défaut	538
40.6.3 Opérateur <code><=></code> 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 <code>==</code>	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 <code>static_cast</code>	551
41.2 <code>const_cast</code>	552
41.3 <code>reinterpret_cast</code>	552
41.4 <code>dynamic_cast</code>	554
41.4.1 Downcasting avec <code>dynamic_cast</code>	557
41.4.2 Sidecasting avec <code>dynamic_cast</code>	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 <code>auto</code>	622
46.2.1 Références explicitement déclarées avec <code>auto</code>	622
46.2.2 Pointeurs explicitement déclarés avec <code>auto</code>	622
46.2.3 Entités déclarées avec <code>auto</code> seul	623
46.3 Différence entre <code>auto</code> et <code>decltype</code>	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 <code>std::move</code>	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 <code>std::forward</code>	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 <code>unique_ptr</code>	645
49.3.1 Création d'un <code>unique_ptr</code>	646
49.3.2 Accès aux données pointées par un <code>unique_ptr</code>	648
49.3.3 Affectation entre <code>unique_ptr</code>	648
49.3.4 Faire pointer un <code>unique_ptr</code> sur un autre objet	649
49.3.5 Permuter des <code>unique_ptr</code>	649
49.3.6 <code>unique_ptr</code> gérant un tableau	649
49.3.7 <code>unique_ptr</code> avec effaceur spécifique	650
49.3.8 La fonction <code>make_unique</code>	652
49.3.9 <code>unique_ptr</code> et transmissions par déplacement	653
49.4 <code>shared_ptr</code>	653
49.4.1 Principe de fonctionnement des <code>shared_ptr</code>	654
49.4.2 Création d'un <code>shared_ptr</code>	655
49.4.3 Faire pointer un <code>shared_ptr</code> sur un autre objet	656
49.4.4 Accès aux données pointées par un <code>shared_ptr</code>	657
49.4.5 Affectation entre <code>shared_ptr</code>	657
49.4.6 Les <code>shared_ptr</code> avec effaceurs spécifiques	657
49.4.7 La fonction <code>make_shared</code>	658
49.4.8 Les <code>shared_ptr</code> et la programmation concurrente	658
49.4.9 Exemple d'utilisation de <code>shared_ptr</code>	659
49.5 <code>weak_ptr</code>	660
49.5.1 Création d'un <code>weak_ptr</code>	662
49.5.2 Dissociation d'un <code>weak_ptr</code> et de l'objet pointé	663
49.5.3 Accès aux données pointées par un <code>weak_ptr</code> grâce à <code>lock</code>	663
49.5.4 Affectation entre <code>weak_ptr</code> , <code>shared_ptr</code> et <code>unique_ptr</code>	665
49.5.5 Un exemple d'utilisation de <code>weak_ptr</code>	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 <code>noexcept</code>	683
51.3.2 Garantie offerte par <code>noexcept</code>	683
51.3.3 <code>noexcept</code> (<i>prédicat_constant</i>)	683
51.3.4 Exemple utilisant le mot-clé <code>noexcept</code>	684
51.3.5 Opérateur <code>noexcept</code>	685
51.3.6 Exemple utilisant l'opérateur <code>noexcept</code>	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 <code>PoLynome</code>	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 <code>IRange</code> 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 <code>atomic<T></code>)	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 <code>jthread</code>	722
53.7.1 Auto-joignabilité	722
53.7.2 Arrêt prématuré d'un <code>jthread</code>	723

54 Programmation concurrente et tâches	727
54.1 Création d'une tâche avec <code>async</code>	727
54.2 Détection de la terminaison d'une tâche	728
54.3 Tâches et exceptions	729
54.4 Fonctionnement des tâches (<code>promise</code> , <code>future</code>)	730
54.4.1 Mécanisme de transmission du résultat d'une tâche	730
55 Objets exécutables asynchrones	733
55.1 Introduction aux <code>packaged_task</code>	733
55.2 Le modèle <code>function</code>	733
55.3 Le modèle <code>packaged_task</code>	734
55.4 <code>packaged_task</code> et exécution concurrente	735
56 Métaprogrammation	737
56.1 Introduction	737
56.1.1 Principe	737
56.1.2 Métaprogrammation et <code>constexpr</code>	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