
目次

序文	17
謝辞	19
第1章 本書について	21
1.1 本書の目的	21
1.2 本書に必要な前提知識	22
1.3 本書のスタイルと構成	22
1.4 本書の利用方法	25
1.5 最新技術	25
1.6 コード例とその他の情報	26
1.7 フィードバック	26
第2章 C++と標準ライブラリの概要	27
2.1 C++と標準ライブラリの歴史	27
2.2 言語の新しい機能	29
2.2.1 テンプレート	29
2.2.2 基本型の明示的な初期化	34
2.2.3 例外処理	34
2.2.4 ネームスペース	36
2.2.5 bool 型	37
2.2.6 explicit キーワード	38
2.2.7 型変換の新しい演算子	38
2.2.8 静的な定数の初期化	40
2.2.9 main()の定義	40
2.3 複雑さとO記法	41
第3章 全体的な概念	43
3.1 std ネームスペース	43
3.2 ヘッダファイル	44
3.3 エラー処理と例外処理	46

3.3.1	標準の例外クラス	46
3.3.2	例外クラスのメンバ	49
3.3.3	標準の例外の送付	50
3.3.4	標準の例外クラスの派生	50
3.4	アロケータ	51

第4章 ユーティリティ 53

4.1	pair	53
4.1.1	make_pair()関数	55
4.1.2	pair の使用例	57
4.2	auto_ptr	57
4.2.1	auto_ptr を使う理由	57
4.2.2	auto_ptr による所有権の譲渡	59
4.2.3	auto_ptr のメンバ	63
4.2.4	auto_ptr の誤用	65
4.2.5	auto_ptr の例	66
4.2.6	auto_ptr の詳細	69
4.3	限界値	75
4.4	補助関数	81
4.4.1	最大値と最小値の処理	81
4.4.2	2つの値の交換	82
4.5	補助的な比較演算子	83
4.6	<cstdint>と<cstdlib>	85
4.6.1	<cstdint>	85
4.6.2	<cstdlib>	86

第5章 STL 87

5.1	STL のコンポーネント	87
5.2	コンテナ	89
5.2.1	シーケンスコンテナ	90
5.2.2	連想コンテナ	95
5.2.3	コンテナアダプタ	96
5.3	反復子	97
5.3.1	連想コンテナの使用例	100
5.3.2	反復子のカテゴリ	106
5.4	アルゴリズム	107
5.4.1	範囲	110

5.4.2	複数の範囲	114
5.5	反復子アダプタ	116
5.5.1	挿入反復子	117
5.5.2	ストリーム反復子	119
5.5.3	逆反復子	121
5.6	アルゴリズムの操作	122
5.6.1	要素の削除	122
5.6.2	操作のアルゴリズムと連想コンテナ	126
5.6.3	アルゴリズムとメンバ関数	127
5.7	ユーザー定義の汎用関数	128
5.8	アルゴリズムの引数としての関数	129
5.8.1	アルゴリズムの引数として関数を使う例	130
5.8.2	叙述関数	132
5.9	関数オブジェクト	134
5.9.1	関数オブジェクトとは	134
5.9.2	事前定義の関数オブジェクト	140
5.10	コンテナの要素	143
5.10.1	コンテナの要素の条件	143
5.10.2	値のセマンティックスと参照のセマンティックス	144
5.11	STLのエラーと例外	146
5.11.1	エラー処理	146
5.11.2	例外処理	148
5.12	STLの拡張	150

第6章 STLのコンテナ 153

6.1	コンテナに共通する機能と演算	153
6.1.1	コンテナの共通機能	153
6.1.2	コンテナの共通演算	154
6.2	vector	157
6.2.1	vectorの機能	158
6.2.2	vectorの演算	159
6.2.3	通常の配列として使うvector	164
6.2.4	例外処理	164
6.2.5	vectorの使用例	165
6.2.6	vector<bool>	166
6.3	deque	168
6.3.1	dequeの機能	169
6.3.2	dequeの演算	170

6.3.3	例外処理	172
6.3.4	deque の使用例	172
6.4	list	174
6.4.1	list の機能	174
6.4.2	list の演算	175
6.4.3	例外処理	179
6.4.4	list の使用例	179
6.5	set と multiset	181
6.5.1	set と multiset の機能	183
6.5.2	set と multiset の演算	184
6.5.3	例外処理	192
6.5.4	set と multiset の使用例	192
6.5.5	ソートの基準の指定(実行時)	196
6.6	map と multimap	198
6.6.1	map と multimap の機能	199
6.6.2	map と multimap の演算	200
6.6.3	連想配列としての map	209
6.6.4	例外処理	210
6.6.5	map と multimap の使用例	210
6.6.6	map、文字列、ソートの基準の指定(実行時)	215
6.7	STL のその他のコンテナ	218
6.7.1	STL コンテナとしての文字列	219
6.7.2	STL コンテナとしての通常の配列	219
6.7.3	ハッシュテーブル	222
6.8	参照のセマンティックスの実装	222
6.9	適切なコンテナの選択	226
6.10	コンテナ型とメンバの詳細	229
6.10.1	型定義	229
6.10.2	生成、コピー、破棄の演算	231
6.10.3	変更を行わない演算	233
6.10.4	代入の演算	237
6.10.5	要素の直接アクセス	238
6.10.6	反復子を生成する演算	240
6.10.7	要素の挿入と削除の演算	241
6.10.8	list の特別なメンバ関数	246
6.10.9	アロケータのサポート	249
6.10.10	STL コンテナにおける例外処理の概要	250

第7章 STLの反復子 253

7.1	反復子のヘッダファイル	253
7.2	反復子のカテゴリ	253
7.2.1	入力反復子	254
7.2.2	出力反復子	255
7.2.3	前方反復子	256
7.2.4	双方向反復子	257
7.2.5	ランダムアクセス反復子	257
7.2.6	vectorの反復子のインクリメントとデクリメント	259
7.3	反復子の補助関数	260
7.3.1	advance()による反復子の移動	260
7.3.2	distance()による反復子の差の計算	262
7.3.3	iter_swap()による反復子の値の交換	264
7.4	反復子アダプタ	265
7.4.1	逆反復子	265
7.4.2	挿入反復子	270
7.4.3	ストリーム反復子	276
7.5	反復子の特性	282
7.5.1	反復子のための汎用関数の作成	283
7.5.2	ユーザー定義の反復子	285

第8章 STLの関数オブジェクト 289

8.1	関数オブジェクトの概念	289
8.1.1	ソートの基準としての関数オブジェクト	290
8.1.2	内部状態を持つ関数オブジェクト	291
8.1.3	for_each()の戻り値	295
8.1.4	叙述関数と関数オブジェクトの違い	297
8.2	事前定義の関数オブジェクト	299
8.2.1	関数アダプタ	300
8.2.2	メンバ関数のための関数アダプタ	301
8.2.3	通常の間数のための関数アダプタ	304
8.2.4	関数アダプタのためのユーザー定義の関数オブジェクト	304
8.3	関数オブジェクトの合成	306
8.3.1	単項関数オブジェクトの関数合成アダプタ	307
8.3.2	二項関数オブジェクトの関数合成アダプタ	311

第9章 STLのアルゴリズム 313

9.1	アルゴリズムのヘッダファイル	313
9.2	アルゴリズムの概要	314
9.2.1	概要	314
9.2.2	アルゴリズムのカテゴリ	315
9.3	補助関数	324
9.4	for_each()のアルゴリズム	325
9.5	変更を行わないアルゴリズム	328
9.5.1	要素のカウント	328
9.5.2	最小と最大	330
9.5.3	要素の検索	331
9.5.4	範囲の比較	344
9.6	変更を行うアルゴリズム	350
9.6.1	要素のコピー	351
9.6.2	要素の変換と結合	354
9.6.3	要素の交換	357
9.6.4	新しい値の代入	358
9.6.5	要素の置換	361
9.7	削除のアルゴリズム	363
9.7.1	指定の値の削除	364
9.7.2	重複の削除	367
9.8	並べ替えのアルゴリズム	371
9.8.1	要素の順序の逆転	371
9.8.2	要素の順序の回転	372
9.8.3	要素の順序の並べ替え	375
9.8.4	要素のシャッフル	377
9.8.5	要素の移動	379
9.9	ソートのアルゴリズム	380
9.9.1	すべての要素のソート	381
9.9.2	部分ソート	384
9.9.3	n 番目の要素に基づくソート	387
9.9.4	ヒープのアルゴリズム	388
9.10	ソートされた範囲を使うアルゴリズム	391
9.10.1	要素の検索	392
9.10.2	要素のマージ	398
9.11	数値演算のアルゴリズム	406
9.11.1	結果を求める処理	406
9.11.2	相対値と絶対値の変換	410

第 10 章 特殊なコンテナ **415**

10.1	stack	415
10.1.1	主なインターフェイス	417
10.1.2	stack の使用例	417
10.1.3	stack の詳細	418
10.1.4	ユーザー定義のスタッククラス	421
10.2	queue	423
10.2.1	主なインターフェイス	424
10.2.2	queue の使用例	425
10.2.3	queue の詳細	426
10.2.4	ユーザー定義のキュークラス	429
10.3	priority_queue	432
10.3.1	主なインターフェイス	433
10.3.2	priority_queue の使用例	434
10.3.3	priority_queue の詳細	435
10.4	bitset	438
10.4.1	bitset の使用例	439
10.4.2	bitset の詳細	441

第 11 章 文字列 **449**

11.1	文字列クラスを使う理由	449
11.1.1	一時的なファイル名の抽出	450
11.1.2	単語の抽出と逆順の出力	454
11.2	文字列クラス	457
11.2.1	文字列型	457
11.2.2	文字列の演算	459
11.2.3	コンストラクタとデストラクタ	461
11.2.4	文字列と C の文字列	461
11.2.5	サイズと容量	463
11.2.6	要素のアクセス	464
11.2.7	比較	466
11.2.8	変更	467
11.2.9	部分文字列と文字列の連結	469
11.2.10	入出力演算子	470
11.2.11	検索	471
11.2.12	npos	472
11.2.13	文字列の反復子	474

11.2.14	国際化	479
11.2.15	パフォーマンス	481
11.2.16	文字列とベクトル	482
11.3	文字列の詳細	482
11.3.1	型定義と静的な値	482
11.3.2	生成、コピー、破棄の演算	484
11.3.3	サイズと容量の演算	486
11.3.4	比較	487
11.3.5	文字のアクセス	489
11.3.6	Cの文字列と文字の配列の生成	490
11.3.7	変更の演算	491
11.3.8	検索	498
11.3.9	部分文字列と文字列の連結	502
11.3.10	入出力関数	503
11.3.11	反復子の生成	504
11.3.12	アロケータのサポート	505
第 12 章	数値	507
12.1	複素数	507
12.1.1	complex の使用例	508
12.1.2	complex の演算	510
12.1.3	complex の詳細	516
12.2	valarray	522
12.2.1	valarray について	523
12.2.2	valarray のサブセット	528
12.2.3	valarray の詳細	542
12.2.4	valarray のサブセットクラスの詳細	549
12.3	数値演算のグローバル関数	553
第 13 章	ストリームクラスによる入出力	557
13.1	入出力ストリームに共通する概念	558
13.1.1	ストリームオブジェクト	558
13.1.2	ストリームクラス	558
13.1.3	グローバルなストリームオブジェクト	559
13.1.4	ストリームの演算子	559
13.1.5	マニピュレータ	560
13.1.6	単純な例	561
13.2	ストリームの基本的なクラスとオブジェクト	561

13.2.1	クラスとクラスの階層	561
13.2.2	グローバルなストリームオブジェクト	565
13.2.3	ヘッダファイル	565
13.3	標準の入出力演算子(<<と>>)	566
13.3.1	出力演算子(<<)	566
13.3.2	入力演算子(>>)	568
13.3.3	特別なデータ型の入出力	568
13.4	ストリームの状態	570
13.4.1	ストリームの状態を表す定数	570
13.4.2	ストリームの状態にアクセスするメンバ関数	572
13.4.3	ストリームの状態とブール値の条件式	573
13.4.4	ストリームの状態と例外	575
13.5	標準入出力関数	579
13.5.1	入力用のメンバ関数	580
13.5.2	出力用のメンバ関数	583
13.5.3	使用例	584
13.6	マニピュレータ	585
13.6.1	マニピュレータのしくみ	585
13.6.2	ユーザー定義のマニピュレータ	587
13.7	書式化	588
13.7.1	書式フラグ	588
13.7.2	ブール値の入出力の書式	590
13.7.3	フィールドの幅、充填文字、位置揃え	591
13.7.4	正の符号と大文字	593
13.7.5	数値の基数	594
13.7.6	浮動小数点数の表記	596
13.7.7	一般的な書式の定義	597
13.8	国際化	598
13.9	ファイルのアクセス	599
13.9.1	ファイルフラグ	603
13.9.2	ランダムアクセス	606
13.9.3	ファイル記述子	608
13.10	入出力ストリームの連結	609
13.10.1	tie()を使う疎結合	609
13.10.2	ストリームバッファを使う密結合	610
13.10.3	標準ストリームのリダイレクト	612
13.10.4	読み取りおよび書き込み用のストリーム	614
13.11	文字列を使うストリームクラス	616

13.11.1	文字列ストリームクラス	616
13.11.2	char*のストリームクラス	620
13.12	ユーザー定義のデータ型の入出力演算子	622
13.12.1	出力演算子の実装	622
13.12.2	入力演算子の実装	624
13.12.3	補助関数を使う入出力	626
13.12.4	書式化なしの関数を使うユーザー定義の演算子	627
13.12.5	ユーザー定義の書式フラグ	629
13.12.6	ユーザー定義の入出力演算子の規約	632
13.13	ストリームバッファクラス	632
13.13.1	ユーザーから見るストリームバッファ	632
13.13.2	ストリームバッファ反復子	634
13.13.3	ユーザー定義のストリームバッファ	637
13.14	パフォーマンスの問題	649
13.14.1	Cの標準ストリームとの同期	649
13.14.2	ストリームバッファにおけるバッファリング	650
13.14.3	ストリームバッファに対する直接アクセス	650

第14章 国際化 653

14.1	さまざまな文字の符号化	654
14.1.1	ワイド文字とマルチバイトのテキスト	654
14.1.2	文字の特性	655
14.1.3	特殊文字の国際化	658
14.2	ロケールの概念	659
14.2.1	ロケールの使用	661
14.2.2	ロケールのファセット	665
14.3	ロケールの詳細	667
14.4	ファセットの詳細	671
14.4.1	数値の書式	672
14.4.2	日付と時刻の書式	675
14.4.3	通貨の書式	679
14.4.4	文字の分類と変換	682
14.4.5	文字列の照合	690
14.4.6	国際化されたメッセージ	692

第15章 アロケータ 693

15.1	アプリケーションのプログラミングにおけるアロケータの使用	693
15.2	ライブラリのプログラミングにおけるアロケータの使用	694

15.3	デフォルトのアロケータ	697
15.4	ユーザー定義のアロケータ	700
15.5	アロケータの詳細	701
15.5.1	型定義	702
15.5.2	演算	703
15.6	未初期化のメモリ用のユーティリティ	705
付録	709
A.1	インターネット上の情報源	709
A.2	参考文献	711
索引	713

第 2 章

C++と標準ライブラリの概要

2.1 C++と標準ライブラリの歴史

C++の標準化は 1989 年に開始され、1997 年末に完了したが、いくつかの動議によって公表は 1998 年の 9 月まで延期された。その結果は ISO(International Organization for Standardization)から約 750 ページのリファレンスマニュアルとして発行された。標準規格の標題は *Information Technology Programming Languages C++*、文書番号は ISO/IEC 14882-1998 で、各国の標準化組織(米国では ANSI)^{*1}から配布されている。

この標準は C++にとって重要な意味を持つ。標準により C++の内容とふるまいが厳密に定義されたため、C++を教えたり、C++をアプリケーションで使ったり、C++を別のプラットフォームに移植するなどの作業が以前より容易になった。また、ユーザーにとっては、C++の実装を選ぶうえで選択枝の幅が広がった。C++の安定性および移植性の向上は、ライブラリやツールの提供や実装に利益をもたらす。C++アプリケーションの開発ではより優れたアプリケーションを時間をかけずに構築できるだけでなく、保守のコストや労力も軽減される。

C++の標準の一部が標準ライブラリである。標準ライブラリは入出力、文字列、コンテナ(データ構造)、アルゴリズム(ソート、検索、マージなど)、数値演算、各種の文字セットを処理するなどの国際化(国際標準だから当然だが)といった機能をサポートする重要なコンポーネントを提供している。

C++の標準化に 10 年近くかかったことや、標準について細かい知識があれば標準化にかけた時間のわりに完全ではないことが不思議に思われる。実は 10 年でも足りなかった

*1 本書の執筆時点では、米国において ANSI の Electronic Standard Store(<http://www.ansi.org/>)を参照)から C++の標準を 18 ドルで購入できる。

訳注: ANSI の Web サイトから標準を入手する方法など、詳しい情報については C++の標準化に関する FAQ(<http://reality.sgi.com/austern/std-c++/faq.html>)にある。

のだ。それでも標準化の歴史と文脈を考えれば、その成果は大きい。完成したものは実用に耐え得る品質ではあるが、完全ではなかった(ただし、世の中に完全なものなどない)。

C++の標準は、莫大な予算と長い時間を利用できる企業によって作られたものではない。標準化組織は、標準の開発者にまったく(またはほとんど)報酬を支払わない。したがって、開発の協力者が、標準に特別な関心を持つ会社に勤務していたのでなければ、標準化の作業は趣味で行われたことになる。それだけのために時間とお金を割くことができる協力者が、ありがたいことに大勢いたのだ。

C++の標準はまったく新規に開発されたわけではない。標準はC++の作者である Bjarne Stroustrup による言語の著書をベースとしている。しかし、標準ライブラリは、書籍または既存のライブラリをベースとしたのではなく、既存のさまざまなクラスが統合された^{*2}ため、品質が均一ではない。たとえば、文字列クラスと STL(データ構造とアルゴリズムのフレームワーク)のように、コンポーネントによって設計方針が異なることがわかる。

文字列クラスは安全で便利なコンポーネントとして設計された。したがって、このクラスのインターフェイスは一目瞭然であり、インターフェイスにおいて数多くのエラーをチェックしている。

STL は各種のデータ構造とさまざまなアルゴリズムを組み合わせて最良のパフォーマンスを得るために設計された。したがって、STL は特別に便利とは言えないうえ、論理エラーの多くはチェックされない。STL の強力なフレームワークと優れたパフォーマンスを活用するには、STL の概念を熟知し、注意深く利用する必要がある。

この2つのコンポーネントが、どちらも同じライブラリに含まれているのである。いくらか協調するように変更されているが、独自の基本的な設計思想は残っている。

ライブラリには、標準化が行われる前から業界標準として存在していたコンポーネントがある。IOStream ライブラリだ。IOStream は 1984 年に開発され、1989 年に部分的に設計を改めて再実装された。すでに多くのプログラムで使われていたため、IOStream の全体的な概念は変更されておらず、下位互換性が保持されている。

言語およびライブラリを含む標準全体は概して盛大な議論の結果であり、世界中の数百人の意見に影響を受けている。たとえば、日本からは国際化に関して重要な支援があった。もちろん、間違いや方向転換もあり、意見は多岐にわたった。そして、1994 年、ようやく標準化の作業が終わりに近づいたと思われたときに STL が組み込まれ、これによりライブラリは根本から変貌した。しかし、標準化を終わらせるために大きな拡張について考えることを事実上停止した。その拡張にどれほどのパワーが秘められているのにもかかわらずだ。結果として、一般的なデータ構造として STL の一部に含めるべきハッシュテーブルは標準に組み込まれなかった。

現在の標準は完全な姿ではない。今後もバグや不整合の修正があると思われるし、5 年も経過すれば標準の新しいバージョンが公表されるだろう。ただし、今後数年は C++ のプ

*2 標準化において新しいライブラリを新規に作成しなかったことを不思議に思うが、標準化の主な役割は、発明や開発ではなく、既存のさまざまな慣例を調停することにあるのだ。

プログラマには標準があり、まったく異なるプラットフォームにも移植可能な、強力なコードを書く機会が与えられているのだ。

2.2 言語の新しい機能

C++の中核言語とライブラリは並行して標準化された。これにより、ライブラリは言語の改良による恩恵を受け、言語はライブラリの実装上の経験からメリットを得る。事実、標準化の作業において、ライブラリはまだ実現されていないC++言語の特別な機能を利用することも多かった。

現在のC++は5年前と同じ言語ではない。C++の進化を追いかけてこなければ、ライブラリが使う新機能を見て驚くはずだ。ここでは、そのような新機能について簡単に説明する。詳細については、C++言語に関する書籍^{*3}を参照してほしい。

本書を執筆している時点では、言語のすべての新機能をすべてのコンパイラが提供しているわけではない。この問題はすぐに改善されると思われる(コンパイラのベンダーのほとんどは標準化の作業に参加していたため)が、現在はライブラリを使う際に制限がある可能性もある。ライブラリの実装で移植性が高いものは、一般に実際に使う環境に機能が存在するかどうかを考慮する(通常は、言語において利用できる機能をチェックするテストプログラムがあり、結果に応じてプリプロセッサのディレクティブを設定する)。本書では、典型的かつ重要な制限については脚注で指摘する。

続いて、C++の標準ライブラリに関する、言語の最も重要な新機能について説明する。

2.2.1 テンプレート

ライブラリのほとんどは、テンプレートとして書かれている。テンプレートをサポートしていなければ、標準ライブラリを使用できない。さらに、ライブラリは特別なテンプレート機能を新しく必要としたのだが、詳細についてはテンプレートを簡単に概説した後で述べる。

テンプレートとは、1つ以上の型を後から指定するように書かれた関数またはクラスのことである。テンプレートを使うときは、明示的または暗黙的に型を引数として渡す。次に、2つの値の大きいほうを返す関数の典型的な例を示す。

```
template <class T>
inline const T& max(const T& a, const T& b)
{
    // a < b の場合には b を、それ以外の場合には a を使う
    return a < b ? b : a;
}
```

*3 訳注: Bjarne Stroustrup 著、株式会社ロングテール/長尾高弘訳『プログラミング言語C++ 第3版』(アスキー、1998年)、Ira Pohl 著、(株)コムサス訳『C++によるオブジェクト指向プログラミング 第2版』(アスキー、1998年)、Scott Meyers 著、吉川邦夫訳『Effective C++ 改訂2版』(アスキー、1998年)など。

最初の行は、Tを任意の型として定義する。型は、この関数を呼び出すときに呼び出し元が指定する。仮引数名には任意の識別子を利用できるが、事実上の規約とは言えないまでも通常はTを使う。型はclassによって分類される^{*4}が、これは必ずしもクラスである必要はない。テンプレートが使う演算さえ提供していれば、どのような型でも利用できる。

同じ原理で、クラスを任意の型に「パラメータ化」することができる。これはコンテナクラスに便利であり、任意の型の要素を処理するコンテナの演算を実装できる。C++の標準ライブラリは、テンプレートとしてコンテナクラスを数多く提供している(第6章「STLのコンテナ」や第10章「特殊なコンテナ」を参照)。テンプレートクラスは、他にもさまざまな理由で使われている。たとえば、文字列クラスは、文字型と文字セットのプロパティに対してパラメータ化されている(第11章「文字列」を参照)。

テンプレートは、任意の型に対して利用できるコードを生成するために一度だけコンパイルされるのではなく、使用する型(または型の組み合わせ)ごとにコンパイルされる。これによって、実際にテンプレートを扱うときにはある重要な問題が発生する。テンプレート関数を、指定する型に応じてコンパイルするには、関数を呼び出すときにその実装が利用可能でなければならない。したがって、今のところ、移植性を持たせつつテンプレートを使うには、ヘッダファイルでインライン関数を使って実装する方法しかない。^{*5}

C++の標準ライブラリの機能を完全に発揮させるには、テンプレートの一般的なサポート機能だけでなく、続いて説明する標準化された新しいテンプレート機能が数多く必要となる。

型以外のテンプレート仮引数

型の仮引数の他に、型以外のテンプレート仮引数を使うことができる。この場合、型以外の仮引数は型の一部であるとみなされる。たとえば、bitset<>という標準クラス(詳細については、第10章「特殊なコンテナ」の10.4「bitset」を参照)では、テンプレート仮引数としてビット数を渡すことができる。次のコードは、32ビットおよび50ビットのビットフィールドをそれぞれ定義する。

```
bitset<32> flags32;    // 32ビットのビットセット
bitset<50> flags50;   // 50ビットのビットセット
```

この2つのbitsetは、使っているテンプレート仮引数が異なるため、型が異なる。したがって、代入や比較を行うことはできない(適切な型変換機能が提供されている場合を除く)。

*4 ここでclassを使ったのは、テンプレートを紹介すると同時に新しいキーワードまで紹介するのを避けたからだ。ただし、現在では、このような場合に使う新しいキーワードとしてtypenameが存在する(31ページの「typename キーワード」を参照)。

*5 テンプレートをヘッダファイルに入れなければならないという問題を回避するために、export キーワードによる「テンプレートコンパイルモデル」が標準に取り入れられた。しかし、著者はまだその実装を見たことがない。

デフォルトのテンプレート仮引数

テンプレートクラスは、デフォルトの仮引数を持つことができる。たとえば、次に示すコードでは、MyClass クラスのオブジェクトを、1 個または 2 個のテンプレート仮引数を持つものとして宣言できる。^{*6}

```
template <class T, class container = vector<T> >
class MyClass;
```

引数を 1 個だけ渡すと、2 番目の引数にはデフォルトの引数が使われる。

```
MyClass<int> x1;    // MyClass<int, vector<int> >に相当する
```

デフォルトのテンプレート引数は、先行引数としても定義されることがある。

typename キーワード

新しく導入された typename キーワードは、続く識別子が型であることを指定する。次に例を示す。

```
template <class T>
class MyClass {
    typename T::SubType * ptr;
    // ...
};
```

typename は、SubType が T クラス型であることを明示するために使われている。したがって、ptr は T::SubType 型のポインタになる。typename がなければ SubType は静的なメンバと解釈されるため、次のコードは、T 型の値である SubType と ptr の乗算になる。

```
T::SubType * ptr
```

SubType が型であることを明示すると、T の場所に指定する型は内部型として SubType を提供しなければならない。たとえば、Q をテンプレート仮引数として次のように使う場合を考える。

```
MyClass<Q> x;
```

この指定は、Q で次のような内部型を定義している場合にのみ有効である。

```
class Q {
    typedef int SubType;
    // ...
};
```

この場合、MyClass<Q>のメンバである ptr は、int 型のポインタとなる。また、SubType はクラスなどの抽象データ型でもよい。

^{*6} 2 つの ' > ' の間に空白を入れなければならないことに注意してほしい。“ >> ”ではシフト演算子と解釈され、文法エラーになる。

```
class Q {
    class SubType;
    // ...
};
```

typename は、たとえ型であることが明確であってもテンプレートの識別子が型であることを示す場合に使う。つまり、C++では原則としてテンプレートの識別子は typename が指定されなければ常に値として解釈される。

また、typename はテンプレートの宣言において class の代わりに使うこともできる。

```
template <typename T> class MyClass;
```

メンバテンプレート

クラスのメンバ関数としてテンプレートが含まれることもある。ただし、メンバテンプレートは仮想関数であってはならないうえ、デフォルトの引数を持つこともできない。

```
class MyClass {
    // ...
    template <class T>
    void f(T);
};
```

このコードでは、MyClass::f() は任意の型の引数をとる一群のメンバ関数として宣言されている。引数として渡す型は、f() が行う演算を提供する任意の型になる。

この機能は、テンプレートクラスのメンバに対して自動的な型変換をサポートするためによく使われる。たとえば、次に示す定義では、assign() の引数の x は、関数が呼び出されるオブジェクトとまったく同じ型でなければならない。

```
template <class T>
class MyClass {
private:
    T value;
public:
    void assign(const MyClass<T>& x) { // xは*thisと同じ型になる
        value = x.value;
    }
    // ...
};
```

assign() の演算のオブジェクトとして別のテンプレート型を使うと、たとえ別の型への自動的な型変換がサポートされていてもエラーとなる。

```
void f()
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d); // OK
    d.assign(i); // エラー(MyClass<double>でなく、MyClass<int>の i が渡される)
}
```

メンバ関数に別のテンプレート型を提供することにより、型一致の規則を緩和させることが可能だ。この方法により、メンバテンプレート関数の引数には、代入可能な任意のテンプレート型を渡すことができる。

```
template <class T>
class MyClass {
private:
    T value;
public:
    template <class X>                // メンバテンプレート
    void assign(const MyClass<X>& x) { // 別のテンプレート型を許可する
        value = x.getValue();
    }
    T getValue() const {
        return value;
    }
    // ...
};

void f()
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d); // OK
    d.assign(i); // OK( int は double に代入できる )
}
```

このとき、assign() の引数の x は、*this の型とは異なることに注意してほしい。したがって、MyClass<>の private や protected のメンバには直接アクセスできない。代わりに、この例のように getValue() などの関数を使う必要がある。

メンバテンプレートの特異な形式として**テンプレートコンストラクタ**がある。テンプレートコンストラクタは、通常、オブジェクトのコピーにおいて暗黙的な型変換が可能になるように提供される。ただし、テンプレートコンストラクタは暗黙的なコピーコンストラクタを隠さないことに注意してほしい。型が完全に一致すれば、暗黙的にコピーコンストラクタが生成され、呼び出される。

```
template <class T>
class MyClass{
public:
    // 暗黙的に型変換を行うコピーコンストラクタ(暗黙的なコピーコンストラクタを隠さない)
    template <class U>
    MyClass(const MyClass<U>& x);
    // ...
};

void f()
{
    MyClass<double> xd;
    // ...
    MyClass<double> xd2(xd); // 組み込みのコピーコンストラクタを呼び出す
    MyClass<int> xi(xd);     // テンプレートコンストラクタを呼び出す
}
```

```
    // ...
}
```

このコードでは、`xd2` と `xd` の型が同じであるため、`xd2` は組み込みのコピーコンストラクタによって初期化される。しかし、`xi` と `xd` の型は異なるため、`xi` はテンプレートコンストラクタを使って初期化される。したがって、テンプレートコンストラクタを書く場合、デフォルトのコピーコンストラクタでは不十分であれば必ずコピーコンストラクタを提供しなければならない。第4章「ユーティリティ」の4.1「`pair`」でもメンバテンプレートの例を紹介する。

ネストしたテンプレートクラス

ネストしたクラスをテンプレートにすることも可能である。

```
template <class T>
class MyClass {
    // ...
    template <class T2>
    class NestedClass;
    // ...
};
```

2.2.2 基本型の明示的な初期化

`explicit` のコンストラクタを引数なしで呼び出すと、基本型を 0 で初期化できる。

```
int i1;           // 値は不定である
int i2 = int();  // 0で初期化される
```

この機能は、どのような型でもデフォルト値を持つテンプレートをコーディングできるように提供された。たとえば、次に示す関数では、基本型の `x` が 0 で初期化されることが保証される。

```
template <class T>
void f()
{
    T x = T();
    // ...
}
```

2.2.3 例外処理

C++の標準ライブラリでは例外処理を使う。この機能を使うと、関数のインターフェイス(引数と戻り値)を変えずに例外の処理が可能である。予想外の問題が発生した場合には、「例外を送出する」ことにより通常の実行を停止できる。

```
class Error;

void f()
{
    // ...
    if (例外の条件) {
        throw Error(); // Error クラスのオブジェクトを生成し、例外として送出する
    }
    // ...
}
```

throw 文により、**スタックのアンwind**という処理が開始される。つまり、ブロックまたは関数から return 文と同様に抜ける。しかし、プログラムがどこかにジャンプするわけではなく、例外によって抜けるブロックで宣言されているすべてのローカルオブジェクトについてそれぞれのデストラクタが呼び出される。スタックのアンwindは、main() を抜けてプログラムが終了するか、catch 句で例外が「捕捉」されて処理されるまで継続する。

```
int main()
{
    try {
        // ...
        f();
        // ...
    }
    catch(const Error&) {
        // ... 例外処理
    }
    // ...
}
```

この例では、try ブロックで発生した Error 型の例外はすべて catch 句で処理される。^{*7} 例外オブジェクトは、通常のクラスまたは基本型で記述される通常のオブジェクトである。したがって、int、文字列、クラスの階層構造に含まれるテンプレートクラスなどで利用できる。ただし、一般的には特別なエラークラス(の階層構造)を設計することが多い。例外オブジェクトの状態を使い、必要な情報をエラーを検出した場所からエラー処理に渡すことができる。

この概念が**エラー処理**ではなく**例外処理**と呼ばれている点に注目しよう。この2つの処理は必ずしも同じではない。たとえば、多くの場合、ユーザーの入力ミスは例外的ではなく一般的に発生するエラーである。そのため、入力ミスは、通常のエラー処理の技法を使ってローカルに処理したほうがよい場合も多い。

例外の指定を記述することにより、関数が返す可能性のある例外のセットを示す。

*7 例外の送出によって関数の呼び出しは終了する。このとき、引数としてオブジェクトを渡して呼び出し元に返すことができるが、これは逆向き(問題が発生した下位レベルから問題を解決または処理する上位レベルへ)の関数のコールバックではない。つまり、例外を処理した後で、例外の発生場所から処理を続行することはできない。その意味で、例外処理はシグナル処理とはまったく異なる。

```
void f() throw(bad_alloc); // f() が送出する例外は bad_alloc だけである
```

関数が例外を送出しないことを示すには、例外のセットを指定しないで宣言する。

```
void f() throw(); // f() は例外を送出しない
```

例外の指定に違反すると、特別なふるまいが発生する。詳細については、第3章「全体的な概念」の3.3「エラー処理と例外処理」を参照してほしい。

C++の標準ライブラリは、標準の例外クラスや `auto_ptr` クラスなど、例外処理のために汎用的な機能をいくつか提供している(詳細については、第3章「全体的な概念」の3.3「エラー処理と例外処理」および第4章「ユーティリティ」の4.2「`auto_ptr`」を参照)

2.2.4 ネームスペース

ライブラリ、モジュール、コンポーネントとして書かれるソフトウェアの量が増えると共に、これらの部品を組み合わせたときに名前の衝突が起こる確率が高くなる。この問題を解決するのがネームスペースである。

ネームスペースは、名前付きのスコープによって複数の識別子をグループ化する。すべての識別子を1個のネームスペースで定義すれば、他のグローバルなシンボルと衝突する危険のあるグローバルな識別子はそのネームスペースそのものの名前だけになる。クラスと同じように、ネームスペース内のシンボルはネームスペースの名前 + `::` 演算子 + 識別子という形式で表す。次に例を示す。

```
// josuttis というネームスペースで識別子を定義する
namespace josuttis {
    class File;
    void myGlobalFunc();
    // ...
}
// ...

// ネームスペース内の識別子を使う
josuttis::File obj;
// ...
josuttis::myGlobalFunc();
```

クラスとは異なり、ネームスペースは別のモジュールでも定義や拡張が自由である。したがって、ネームスペースを使ってモジュール、ライブラリ、コンポーネントを複数のファイルで定義できる。ネームスペースが定義するのは、物理モジュールではなく論理モジュールである(UML などのモデル記法ではモジュールは **パッケージ** とも呼ばれる)

関数の引数の型がネームスペースで1つ以上定義されていれば、ネームスペースでその関数を修飾する必要はない。この規則は Koenig **参照** と呼ばれている。次に例を示す。

```
// josuttis というネームスペースで識別子を定義する
namespace josuttis {
    class File;
```

```
    void myGlobalFunc(const File&);  
    // ...  
}  
// ...  
  
josuttis::File obj;  
// ...  
myGlobalFunc(obj);    // OK - josuttis::myGlobalFunc() が参照される
```

using の宣言により、他のネームスペースのスコープを繰り返し修飾することを回避できる。

```
using josuttis::File;
```

このように宣言すると、現在のスコープでは File が josuttis::File と同じ意味を持つ。

using を使うと、指定したネームスペースのすべての名前が利用可能になる。ネームスペースの外側で宣言されている名前を使用できるようになるわけだが、同時に名前の衝突が発生する可能性もある。

```
using namespace josuttis;
```

このように宣言すると、File と myGlobalFunc() は現在のスコープにおいてグローバルな識別子になる。グローバルなスコープにも File や myGlobalFunc() という識別子があり、ユーザーがその名前を修飾子を付けずに使うと、コンパイラは識別子があいまいであることを警告する。

ヘッダファイル、モジュール、ライブラリの中など、コンテキストが明確でない場所では決して using を使ってはならない。これは using によりネームスペースの識別子のスコープが変わる可能性があるからだ。コードを別のモジュールでインクルードしたり、利用すると、予想外のふるまいが発生することもある。実際、ヘッダファイルで using を使うのはまったくの設計ミスである。

C++の標準ライブラリでは、すべての識別子を std というネームスペースで定義している。詳細については、第3章「全体的な概念」の3.1「std ネームスペース」を参照してほしい。

2.2.5 bool型

ブール値のサポートを強化するために、bool 型が導入されている。bool を使うことによってコードの読みやすさが増し、ブール値のふるまいのオーバーロードが可能になった。また、ブール値として true と false というリテラルが導入されたほか、整数値と相互に自動的に型変換が行われ、false は 0 という値に等しく、それ以外の値は true となる。

2.2.6 explicitキーワード

`explicit` というキーワードを使うと、1 個の引数を受け取るコンストラクタが自動的に型変換を定義することを禁止できる。この機能は、コンストラクタの引数として初期サイズを渡すコレクションクラスで一般的に必要となる。たとえば、スタックの初期サイズを引数とするコンストラクタは、次のように宣言される。

```
class Stack {
    explicit Stack(int size);    // 初期サイズを指定してスタックを生成する
    // ...
};
```

この例では `explicit` が非常に重要な意味を持つ。`explicit` がなければ、コンストラクタは `int` から `Stack` への自動的に型変換を定義するため、`int` を `Stack` に代入できるようになってしまう。

```
Stack s;
// ...
s = 40;    // おっと、40 個の要素を持つ Stack が生成され、s に代入される
```

自動的に型変換により、40 という整数値が 40 個の要素を持つスタックに変換され、`s` に代入される。これはおそらく本来の意図とは異なる処理である。`int` のコンストラクタを `explicit` で宣言すれば、このような代入処理はコンパイル時にエラーとなる。

`explicit` を使うと、代入構文によって自動的に型変換を行う初期化処理も禁止される。

```
Stack s1(40);    // OK
Stack s2 = 40;    // エラー
```

次に示すコードを比較してみよう。

```
X x;
Y y(x);    // 明示的な変換
```

この処理は、次に示す処理とは微妙に異なる。

```
X x;
Y y = x;    // 暗黙的な変換
```

最初のコードでは `Y` 型の新しいオブジェクトを `X` 型から明示的に変換を行って生成する。一方、その次のコードでは、暗黙的に変換を行って `Y` 型のオブジェクトを生成する。

2.2.7 型変換の新しい演算子

1 個の引数に対する明示的な型変換の意味を明確に示すことができるように、次の 4 種類の新しい演算子が導入された。

1. `static_cast`

この演算子は値を論理的に変換する。一時的なオブジェクトを生成し、変換される値によって初期化するとみなすことができる。変換は、組み込みの変換規則や変換の演算の定義により型変換が定義されている場合に限られる。次に例を示す。

```
float x;
// ...
cout << static_cast<int>(x);      // xをintとして出力する
// ...
f(static_cast<string>("hello")); // char*ではなくstringを受け取るf()を呼び出す
```

2. `dynamic_cast`

この演算子を使うと、ポリモーフィックな型を実際の静的な型にダウンキャストできる。実行時には唯一このキャストがチェックされる。したがって、ポリモーフィックな値の型チェックに利用可能である。次に例を示す。

```
class Car;          // 車を表す抽象基本クラス(1つ以上の仮想関数を持つ)

class Cabriolet : public Car { // カブリオレを表すクラス
    // ...
};

class Limousine : public Car { // リムジンを表すクラス
    // ...
};

void f(Car* cp)
{
    Cabriolet* p = dynamic_cast<Cabriolet*>(cp);
    if (p == NULL) {
        // pはCabriolet型のオブジェクトを参照しない
        // ...
    }
}
```

この例の `f()` は、実際には静的な `Cabriolet` 型であるオブジェクトについて特別な処理を行う。引数として参照が渡され、型変換が失敗すると、`dynamic_cast` は `bad_cast` という例外を送出する(`bad_cast` については、第3章「全体的な概念」を参照)。しかし、設計という側面から見ると、ポリモーフィックな型を使うプログラミングでは型に依存する文を使わない設計のほうが優れている。

3. `const_cast`

この演算子は型に対して `const` の指定を追加または削除する。また、`volatile` の修飾子を削除することもできる。それ以外の型の変更は許されていない。

4. `reinterpret_cast`

この演算子のふるまいは実装に応じて定義される。ビットの再解釈を行うこともあるが、必須ではない。この型キャストは通常移植できない。

これらの演算子はかっこを使う古いキャストの技法に代わるもので、変換の意図を明確にするという利点がある。型変換は、`dynamic_cast` 以外はかっこを使う古いキャスト形式に置き換えられるが、かっこの型キャストでは型変換を行う理由を厳密に示すことができない。新しい演算子を使えば、コンパイラは変換の理由について詳細な情報を得ることになり、許可されていない変換を試みるコードにエラーを通知できる。

各演算子は、1つの引数を持つコンストラクタ用に提供されている。次に例を示す。

```
static_cast<Fraction>(15,100)    // おっと、Fraction(100) が生成される
```

しかし、これでは意図したとおりのふるまいは行われぬ。このコードは、分子が15で分母が100の一時的な分数を初期化するのではなく、100という値を1つだけ持つ一時的な分数を初期化する。この場合、カンマは引数を区切る記号ではなく、「2つの式を組み合わせる1つの式とみなし、2番目の式を値として使う演算子」とみなされる。15と100という値を使って分数に「変換する」には、次のように書くのが正解である。

```
Fraction(15,100)    // 問題なし(Fraction(15,100) が生成される)
```

2.2.8 静的な定数の初期化

クラスの構造において静的なメンバである整数型の定数値の初期化が可能になった。これは、クラスで初期化してから利用するような定数に便利である。次に例を示す。

```
class MyClass {
    static const int num = 100;
    int elems[num];
    // ...
};
```

静的な定数を初期化するには、クラスの定義において領域を確保する必要があることに注意してほしい。

```
const int MyClass::num;    // 初期化されない
```

2.2.9 `main()` の定義

中核言語において、重要だが、誤解されることが多い領域についても明確にしておきたい。つまり、唯一移植性を持つ `main()` のことだ。C++の標準によれば、移植性を持つ `main()` の定義には2種類ある。

```
int main()
{
    // ...
}
```

```
int main(int argc, char* argv[])
{
    // ...
}
```

2番目の定義の `argv` (コマンドライン引数の配列)は `char**`と定義してもよい。戻り値として `int` 型を定義しなければならないのは、暗黙的な `int` 型の戻り値が推奨されなくなったからだ。

`main()` を `return` 文で終了させることはできるが、必須ではない。Cとは異なり、C++は `main()` の最後に暗黙的に次の文を定義する。

```
return 0;
```

つまり、`return` 文なしに `main()` を終了させるプログラムはすべて成功したことになる(0以外の値を返すと何らかの失敗を意味する)。このため、本書のコード例では `main()` の最後に `return` 文を書いていない。コンパイラには、`return` 文のないプログラムに対して警告メッセージを出したり、エラーとみなすものがあるが、これは標準よりはるか昔の話である。

2.3 複雑さとO記法

C++の標準ライブラリの一部(特に STL)では、アルゴリズムとメンバ関数のパフォーマンスが慎重に検討され、標準ではその複雑さを示す指標が必要となった。コンピュータ科学者はアルゴリズムの相対的な複雑さを比較するのに特別な記法を使う。この指標により、アルゴリズムの相対的な実行時間をすばやく分類したり、アルゴリズムの品質を比較することができる。この指標は、**O記法**と呼ばれる。

O記法は、アルゴリズムの実行時間を、入力値に n というサイズを指定された関数として表現する。たとえば、実行時間が要素の数に対してほぼ線形に増大する(入力が倍になると実行時間も倍になる)場合、その複雑さは $O(n)$ で表される。実行時間が入力に依存しない場合、複雑さは $O(1)$ になる。表 2-1 に、複雑さを示す一般的な値と O記法による表現を示す。

表 2-1 複雑さを表す値

種類	記法	説明
一定	$O(1)$	実行時間は要素数に依存しない。
対数的	$O(\log(n))$	実行時間は要素数に対して対数的に増大する。
線形	$O(n)$	実行時間は要素数に対して線形に増大する(実行時間と要素数が比例して増大する)。
$n \cdot \log n$	$O(n * \log(n))$	実行時間は線形の複雑さと対数的な複雑さの積として増大する。
平方的	$O(n^2)$	実行時間は要素数に対して平方的(二次関数的)に増大する。

O記法では、小さな指数(定数項など)は隠されるということが意味を持つ。また、アルゴリズムの実行時間には関係しない。O記法では、線形アルゴリズムなら種類に関係なく同程度に評価される。同じ線形アルゴリズムでも定数項が大きすぎるため、実際には、定数項が小さい対数的アルゴリズムのほうが望ましい場合も考えられる。これはO記法に対する正当な批判である。O記法は単なる経験則にすぎないことに注意してほしい。複雑さが最適な値を示すアルゴリズムが必ずしも最良とは限らない。

表2-2に、各アルゴリズムの複雑さと要素数の例を示す。要素数の増加に対して実行速度がどれだけ速くなるかを感覚的につかんでほしい。このように要素数が少ない場合には、実行時間はそれほど変わらない。また、O記法によって隠される定数項が大きな影響を及ぼすこともある。ただし、要素数がさらに増えると、実行時間の違いが大きくなるため、定数項の影響は少なくなる。複雑さを考慮するときには、大まかな目安として考える必要がある。

表2-2 複雑さと要素数による実行時間の違い

複雑さ		要素の数						
種類	記法	1	2	5	10	50	100	1,000
一定	$O(1)$	1	1	1	1	1	1	1
対数的	$O(\log(n))$	1	2	3	4	6	7	10
線形	$O(n)$	1	2	5	10	50	100	1,000
n-log-n	$O(n * \log(n))$	1	4	15	40	300	700	10,000
平方的	$O(n^2)$	1	4	25	100	2,500	10,000	1,000,000

C++のリファレンスマニュアルによっては、複雑さの定義に**償却された値**(amortized)と書かれているものがある。これは、演算が「長い目で見れば」記述どおりにふるまうという意味である。1回の実行では仕様より長くかかる可能性がある。たとえば、動的な配列に要素を追加する場合、配列に要素をもう1つ追加するのに十分なメモリが残されているかどうかは実行時間に影響を与える。十分なメモリがあれば、最後に要素を新しく追加する時間は常に同じであるため、複雑さは一定である。しかし、十分なメモリがなければ、要素の実際の数に応じて新しいメモリを割り当ててすべての要素をコピーしなければならないため、複雑さは線形になる。ただし、このように配列が再配置されることはどちらかと言えばまれであるため、この演算は十分に長い間繰り返し実行すれば各演算の複雑さが一定であるかのようにふるまう。したがって、挿入の演算の複雑さは一定の時間に「償却される」のだ。