序文

プログラミングとは理解することだ
——*Kristen Nygaard*

C++を使うのは、今まで以上に楽しくなっていると思う。ここ数年で、C++の設計、プログラミングに対するサポート能力は飛躍的に向上し、C++を使う上で役に立つ新しいテクニックも無数に開発された。しかし、C++はただ面白い**だけ**ではない。実際に仕事をしている普通のプログラマたちが、あらゆる種類、あらゆる規模のプロジェクトで、生産性、保守性、柔軟性、品質の大幅な改善を達成しているのである。C++は、私がもともとこの言語で実現したいと思っていたことの大半を達成するとともに、私が夢にも思わなかったような仕事でも成功を収めている。

本書は、標準 C++[†]を紹介し、C++がサポートする主要なプログラミング、設計テクニックを示す。標準 C++は、本書の第 1 版が世に出したバージョンの C++よりもはるかに強力で洗練された言語である。名前空間、例外、テンプレート、実行時型識別などの新しい言語機能が導入されたために、さまざまなテクニックを従来よりもより直接的に使えるようになったし、標準ライブラリが整備されたために、プログラマは裸の言語よりもはるかに高いレベルから仕事を始められるようになった。

本書第 2 版に書かれている内容のうち、第 1 版から流用されているものは全体の 1/3 ほどである。この第 3 版は、それ以上に大きな規模のリライトによって完成した。本書は、もっとも熟練した C++プログラマにも、なにがしかのものを与えられるはずである。同時に、本書は、従来の版よりも初心者がアプローチしやすいものに仕上がっている。このようなことが可能になったのは、C++が爆発的に利用され、膨大な経験が蓄積されたためである。

大規模な標準ライブラリが定義されたことにより、C++のコンセプトの提示方法は変わった。以前の版では、特定の処理系からは独立した形で C++を提示していたし、チュートリアルの章では、言語要素を定義してから使うという形の説明になるように、“ ボトムアップ ”方式で言語要素、コンセプトを提示していた。しかし、ライブラリがうまく設計されていれば、その実装のディティールを理解するよりも、使う方がはるかに簡単である。そのため、現実的で面白いサンプルを提供すれば、読者が標準ライブラリの内部の仕組みを理解するよりも前に、標準ライブラリを使いこなすことは可能である。そして、標準ライブラリ自体も、プログラミング例と設計テクニックの豊穣な源泉である。

† ISO/IEC 14882, Standard for C++ Programming Language

本書は、C++言語のすべての主要機能と標準ライブラリを示す。本書の構成は、言語とライブラリの機能を中心としたものになっているが、機能は使うときの文脈に沿って説明されている。つまり、本書が重点を置いているのは、言語自体ではなく、設計、プログラミングのツールとしての言語である。本書は、C++を効果的に活用するための重要テクニックを具体的に示し、C++の熟達に必要な基本コンセプトを教えていく。専門的な機能を説明している箇所を除き、サンプルはシステムソフトウェアの領域から取られている。姉妹版の"The Annotated C++ Language Standard"には、理解を助けるためのコメントとともに、言語の完全な定義が含まれている。

本書の第1の目的は、C++の機能が主要なプログラミングテクニックをどのようにサポートしているかについての理解を助けることである。主としてサンプルのコードをコピーし、他の言語のプログラミングスタイルをエミュレートすることによってコードを何とか動かしているような地点からはるかに遠いところに読者を連れていきたい。言語に熟達するためには、言語機能の背後にある考え方を充分理解することが必要不可欠である。本書に書かれている内容に処理系のマニュアルの内容を補えば、大規模プロジェクトを充分に完成に導けるだろう。私の望みは、読者が本書によって新しい洞察力を獲得し、より優れたプログラマ、デザイナになることである。

謝辞

第1、第2版の謝辞のセクションで言及した人々に加え、第3版の草稿にコメントを加えてくれた、Matt Austern、Hans Boehm、Don Caldwell、Lawrence Crowl、Alan Feuer、Andrew Forrest、David Gay、Tim Griffin、Peter Juhl、Brian Kernighan、Andrew Koenig、Mike Mowbray、Rob Murray、Lee Nackman、Joseph Newcomer、Alex Stepanov、David Vandevoorde、Peter Weinberger、Chris Van Wyk に感謝の意を捧げたい。彼らの助言と忠告がなければ、本書はもっとわかりにくく、エラーの数が多く、完成度のはるかに低いものになっていただろうし、おそらくもう少し短いものになっていただろう。

C++を現在の姿に鍛え上げるために膨大な量の建設的な作業を成し遂げた C++標準委員会のボランティアたちにも感謝の意を捧げたい。一部の名前だけを抜き出すのは若干不公平だろうが、誰も取り上げないのでは、もっと不公平になってしまう。そこで、C++とその標準ライブラリの何らかの部分について私と直接共同作業に携わった Mike Ball、Dag Brück、Sean Corfield、Ted Glodstein、Kim Knuttila、Andrew Koenig、Josée Lajoie、Dmitry Lenkov、Nathan Myers、Martin O'Riordan、Tom Plum、Jonathan Shopiro、John Spicer、Jerry Schwarz、Alex Stepanov、Mike Vilot の名前を特に掲げて感謝の意を表わしたい。

Murray Hill, New Jersey

Bjarne Stroustrup

第2版への序文

道は永遠に続く
——*Bilbo Baggins*

本書第 1 版でお約束したとおり、C++はユーザーのニーズに応えるために発展を遂げてきた。この発展を導いてきたのは、さまざまな応用分野で働いているさまざまな経歴を持ったユーザーの経験である。C++ユーザーコミュニティは、本書第 1 版が出版されてからの 6 年の間に、100 倍にも拡大した。経験によって多くの教訓が得られ、多くのテクニックが発見、確認されてきた。本書には、それらの経験の一部が反映されている。

この 6 年間に加えられてきた言語拡張の主目的は、一般的にデータ抽象、オブジェクト指向プログラミング全般のための言語として C++を整備すること、特にユーザー定義型の高品質ライブラリを書くためのツールとして C++を改良することだった。“ 高品質ライブラリ ”とは、ユーザーに便利で安全で効率的に使える 1 つ以上のクラスの形でコンセプトを提供するライブラリのことである。この文脈において、**安全**とは、クラスが、ライブラリの実装者と利用者の間に、型の安全が保証されたインターフェイスを提供することを意味する。**効率**とは、クラスを使ったときに、手書きの C コードと比べて、実行時間、空間のオーバーヘッドが過大にならないようにすることである。

本書は、C++言語を完全な形で示す。第 1 章から第 10 章まではチュートリアル的な入門の章である。第 11 章から第 13 章までは、設計とソフトウェア開発の問題を取り上げる。そして、最後に完全な C++リファレンスマニュアルを添付している。当然ながら、第 1 版以降の追加機能、決定は、説明の一部として完全に統合されている。そのなかには、改良された多重定義解決、メモリ管理機能、アクセス制御メカニズム、型の安全を保証したリンケージ、*const* 及び *static* メンバ関数、抽象クラス、多重継承、テンプレート、例外処理が含まれる。

C++は、汎用プログラミング言語である。主要な応用分野は広い意味でのシステムプログラミングとなっているが、このラベルではカバーできないさまざまな応用分野でも、C++は成功している。C++の処理系は、もっともつつましいマイクロコンピュータの一部からもっとも大規模なスーパーコンピュータまで、ほぼすべてのオペレーティングシステムを対象として作られている。そこで、本書は特定の処理系、プログラミング環境、ライブラリを説明するのではなく、C++言語自体を説明する。

本書には、便利ではあるものの、“ オモチャ ”に分類すべき多くのクラスのサンプルが含まれている。このようなスタイルで説明を進めると、本格的なプログラムのクラスを使ったときと比べて、一般原則や便利なテクニックがより鮮明に浮かび上がってくる。本格的

なプログラムでは、これらのものはディティールのなかに埋もれてしまうのである。本書に含まれているリスト、配列、文字列、行列、グラフィックスクラス、連想配列などの役に立つクラスの大半については、"頑丈"で"金メッキ付き"のバージョンが市販、非市販などのさまざまな形で流通している。これらの"産業用の強度を持つ"クラス、ライブラリの多くは、本書のオモチャバージョンの直接、間接の子孫である。

第 2 版では、第 1 版よりもチュートリアル的な側面の比重を高くした。しかし、ベテランプログラマを対象としたストレートな記述スタイルを保ち、彼らの知性や経験を侮辱しないよう、充分に努力したつもりである。また、言語機能とその目先の使い方の先にある情報に対する需要に応えて、設計問題についての議論を大幅に拡充した。技術的なディティールと精度も向上させた。特に、リファレンスマニュアルは、この分野における長年の仕事を反映したものになっている。ほとんどのプログラマにとって、再読の価値のある深さを備えた本を書くことを目指した。言い換えれば、本書は C++言語とその基本原則、それを適用するために必要な主要テクニックを説明している。楽しんでいただきたい。

謝辞

第 1 版の謝辞のセクションで言及した人々に加え、第 2 版の草稿にコメントしてくれた、Al Aho、Steve Buroff、Jim Coplien、Ted Goldstein、Tony Hansen、Lorraine Juhl、Peter Juhl、Brian Kernighan、Andrew Koenig、Bill Leggett、Warren Montgomery、Mike Mowbray、Rob Murray、Jonathan Shopiro、Mike Vilot、Peter Weinberger に感謝の意を捧げたい。1985 年から 1991 年までの C++の発展には多くの人々が影響を与えた。ここで触れられるのは、Andrew Koenig、Brian Kernighan、Doug McIlroy、Jonathan Shopiro といったごく一部の人々の名前だけである。また、リファレンスマニュアル案の"外部レビュー"の多くの参加者や、X3J16 の初年度で苦しみを分かち合った人々にも謝意を捧げたい。

Murray Hill, New Jersey

Bjarne Stroustrup

第1版への序文

言語は思考の方法を形成し、
思考できることの限界を定義する
——*B. L. Whorf*

C++は、本格的なプログラマにとってプログラミングがもっと楽しい仕事になることを目指して設計された汎用プログラミング言語である。わずかなディティールを除けば、C++はC言語のスーパーセットである。C++は、Cが提供する機能に加えて、新しい型を定義するための柔軟で効率の良い機能を提供している。プログラマは、アプリケーションのコンセプトに密接に結びついた新しい型を定義することによって、アプリケーションを管理可能な部品に分割することができる。プログラム構築のためのこのテクニックは、**データ抽象**: *data abstraction* と呼ばれることが多い。ユーザー定義型のオブジェクトには、型情報が含まれている。このようなオブジェクトは、コンパイル時に型を判定できないような文脈でも便利、かつ安全に使うことができる。このようなタイプのオブジェクトを使ったプログラムは、**オブジェクトベース**: *object based* のプログラムと呼ばれることが多い。これらのテクニックをうまく使うと、短くて理解しやすく、維持しやすいプログラムを作れる。

C++のキーコンセプトは、**クラス**: *class* である。クラスはユーザー定義型であり、データの隠蔽、データの確実な初期設定、ユーザー定義型のための暗黙の型変換、動的な型付け、ユーザー制御のメモリ管理、演算子多重定義メカニズムを提供する。C++は、型チェックとモジュール性の表現についてはCよりもはるかに優れた機能を提供する。また、シンボル定数、関数のインライン置換、デフォルト関数引数、関数名の多重定義、自由記憶領域管理演算子、リファレンス型など、クラスとは直接関係のない改良も含んでいる。その一方で、ハードウェアの基本オブジェクト（ビット、バイト、ワード、アドレスなど）を効率よく扱えるCの能力を維持している。そのため、心地よい効率の良さでユーザー定義型を実装できる。

C++とその標準ライブラリは、移植性を考えて設計されている。現在の実装は、Cをサポートするほとんどのシステムで動作するはずである。C++プログラムはCライブラリを使え、Cプログラミングをサポートするほとんどのツールは C++でも使える。

本書は、本格的なプログラマが C++言語を学習し、一定以上の規模を持つプロジェクトで C++言語を使えるようにすることを主目的として書かれている。本書は、C++を完全な形で説明し、多くの完成したサンプルプログラムとそれよりも多くのプログラム片を示す。

謝辞

多くの友人、同僚が継続的に利用し、提案し、建設的な批評を加えてくれなければ、C++言語はここまで成熟しなかっただろう。特に、Tom Cargill、Jim Coplien、Stu Feldman、Sandy Fraser、Steve Johnson、Brian Kernighan、Bart Locanthi、Doug McILroy、Dennis Ritchie、Larry Rosler、Jerry Schwart、Jon Shopiro は、この言語を開発するに当たっての重要なアイディアを提供してくれた。Dave Presotto は、ストリーム入出力ライブラリの現在の実装を書いてくれた。

このほかにも数百人もの人々が、改良案、障害報告、コンパイラエラーなどを私に送ってくれて C++とそのコンパイラの開発に貢献している。そのなかのごく一部だが、Gary Bishop、Andrew Hume、Tom Karzes、Victor Milenkovic、Rob Murray、Leonie Rose、Brian Schmult、Gary Walker の名前を挙げておきたい。

また、多くの人々が本書の製作を助けてくれた。特に、Jon Bentley、Laura Eaves、Brian Kernighan、Ted Kowalski、Steve Mahaney、Jon Shopiro、1985 年 6 月 26 日から 27 日にオハイオ州コロンバスのベル研究所で開催された C++コースの参加者に謝意を捧げたい。

Murray Hill, New Jersey

Bjarne Stroustrup

イントロダクション

このパートでは、プログラミング言語 C++とその標準ライブラリの主要概念、機能の概要を示す。また、本書全体の概要を示し、本書では言語の機能と使い方をどのように扱っているかも説明する。さらに、C++とその設計、利用方法の背景となる知識についても説明する。

第 1 章 読者へのメッセージ
第 2 章 C++ひとめぐり
第 3 章 標準ライブラリひとめぐり

Marcus、君は私に多くのものを与えてくれた。今度は私が君に良きアドバイスを贈ろう。いつも良き Marcus Cocoza であろうとするな。君は、Marcus Cocoza のことを考える余り、彼の奴隷、虜囚となってしまった。君は何をするにも、まずそれが Marcus Cocoza の幸福と名誉に与える影響を考えていた。君はいつも Marcus が愚かに見えたり退屈に感じられることを恐れていた。しかし、それは本当に大事なことなのだろうか？
世界中の人々が愚かなことをしている。君ももっと楽になった方がいい。そうすれば、君の気持ちも再び明るいものになるだろう。これからの君は、君一人ではなく、考えられる限り多くの人の立場に立つべきだ。

——Karen Blixen
(Isak Dinesen の名で発表された“ Seven Gothic Tales ”のなかの“ The Dreamers ”Rondom House,Inc.
Copyright, Isac Dinesen, 1934 renewed 1961)

第 1 章

読者へのメッセージ

せいうちは言った。「多くのことを語るべきときが来た」
――*L.Carroll*

本書の構造◆C++の学習方法◆C++の設計◆効率と構造◆設計思想についてのコメント◆歴史的背景◆C++の用途◆C と C++◆C プログラマのためのヒント◆C++プログラミングについての考え方◆アドバイス◆参考文献

1.1 本書の構造

本書は 6 部構成になっている。

イントロダクション:	第 1 章から第 3 章までは、C++言語とそれがサポートするプログラミングスタイル、C++標準ライブラリの概要を示す。
第 1 部:	第 4 章から第 9 章までは、C++の組み込みデータ型とプログラム構築のための基本機能を紹介する
第 2 部:	第 10 章から第 15 章までは、C++を使ったオブジェクト指向プログラミング、ジェネリックプログラミングの方法を紹介する。
第 3 部:	第 16 章から第 22 章までは、C++標準ライブラリを紹介する。
第 4 部:	第 23 章から第 25 章までは、ソフトウェアの設計、開発の問題を取り上げる。
付録:	付録 A から付録 C までは、プログラミング言語としての詳細を示す。

第 1 章は、本書の概要と使い方のヒントを示すとともに、C++とその使い方を理解するための背景となる知識を示す。全体に目を通し、面白そうなところを読み、本書のほかの部分を読んだあとでまた読み直すとよいだろう。

第 2 章、第 3 章は、C++プログラミング言語と標準ライブラリの主要概念、機能を概観

する。この 2 章では、完全な C++言語を使えばどのようなことが表現できるかを示すことによって、基本概念や言語の基本機能について読者にじっくり考えていただきたいと思っている。少なくとも、これらの章を読めば、C++がただの C ではなく、本書の第 1、2 版のころの形からも大きく変わっていることを理解していただけるだろう。第 2 章は、高いレベルから C++を見ていく。ここで焦点を当てるのは、データ抽象、オブジェクト指向プログラミング、ジェネリックプログラミングをサポートする言語機能である。第 3 章は、標準ライブラリの基本原則と主要な機能を示す。標準ライブラリについてこの段階で示しているので、以降の章では、より低い水準の組み込み機能を直接使うのではなく、標準ライブラリの機能を使っていく。

これらの 3 章は、本書全体を貫く記述方法の生きた例にもなっている。本書では、テクニックや機能を直接的かつリアルに説明するために、最初にコンセプトを簡単に示してからあとで詳しく論じることがよくある。こうすれば、あるトピックを一般的に扱う前に、より具体的な例を示すことができる。つまり、本書は、具体的なものから抽象的なものに進むという学習効果の高いことが実証されている方法を使って構成されている（あとから考えれば、抽象的な概念の方が単純でわかりやすく感じられるものだが）。

第 1 部は、C++の機能のうち、従来、C や Pascal で使われてきたプログラミングスタイルをサポートする部分を紹介する。C++プログラムの基本データ型、式、制御構造はここで説明する。また、名前空間、ソースファイル、例外処理がサポートするモジュール化についても論じる。第 1 部では、読者が基本的なプログラミングの概念を理解していることを前提として話を進める。たとえば、再帰や反復を表現するための C++の機能については説明するが、これらの概念がどのように役に立つかについてはあまり説明しない。

第 2 部では、C++の機能のうち、新しい型を定義して使う部分を説明する。ここでは、具象クラスと抽象クラス（インターフェイス）（第 10 章、第 12 章）、演算子の多重定義（第 11 章）、クラス階層の使い方と多相化（第 12 章、第 15 章）を取り上げる。また、第 13 章では、型と関数のファミリを定義する機能であるテンプレートを論じる。ここでは、リストなどのコンテナを提供し、ジェネリックプログラミングをサポートするための基本テクニックを示す。例外処理機能を紹介する第 14 章では、エラー処理のテクニックとフォルトトレランスを実現するための戦略について論じる。第 2 部では、読者がオブジェクト指向プログラミングとジェネリックプログラミングについての知識をあまり持たず、C++がサポートする主要な抽象化テクニックの説明を求めていることを前提として話を進める。つまり、単に抽象化テクニックをサポートする言語機能を紹介するだけではなく、テクニック自体についても説明する。この議論は、第 4 部でさらに深く掘り下げていく。

第 3 部は、C++標準ライブラリを示す。この部分の目標は、ライブラリの使い方の理解を深め、一般的な設計、プログラミングテクニックを具体的に展開し、ライブラリの拡張方法を示すことである。標準ライブラリは、コンテナ（list、vector、map など。第 16 章、第 17 章）、標準アルゴリズム（sort、find、merge など。第 18 章、第 19 章）、文字列（第 20 章）、入出力（第 21 章）、数値演算サポート（第 22 章）を提供する。

第 4 部は、大規模システムの設計、実装に C++を使うときの問題を論じる。第 23 章は、設計、管理の問題に焦点を絞り、第 24 章でそれら設計上の問題と C++プログラミング言

語の関係を論じる。第 25 章は、設計時のクラスの使い方について取り上げる。

付録 A は、コメント付きの C++の文法規則である。付録 B は、C と C++の関係、標準 C++（ISO C++、あるいは ANSI C++とも呼ばれる）と以前のバージョンの C++の関係を取り上げる。付録 C は、言語の専門的な問題を取り上げる。

1.1.1 サンプルと参照箇所の凡例

本書は、アルゴリズムの書き方よりもプログラムの構成に重点を置く。そのため、巧妙なアルゴリズムや理解しにくいアルゴリズムは避けられている。一般に、言語定義の個々の側面やプログラムの構造についてのポイントを説明するときには、簡単なアルゴリズムの方が適している。たとえば、現実のコードではクイックソートの方が適している箇所でも、シェルソートを使っている。より優れたアルゴリズムを使った再実装は、練習問題になっていることが多い。そして、現実のコードでは、本書で言語機能を説明するために使ったコードよりもライブラリ関数を呼び出した方が一般によい。

教科書のサンプルコードは、必然的にソフトウェア開発を簡単に見せてしまうものである。サンプルを明確かつ単純にすることによって、規模がもたらす複雑さは見えなくなる。プログラミングとプログラミング言語の現実のイメージをつかむためには、現実的な規模のプログラムを書いてみる以外に方法はないと思う。しかし、本書では、言語の機能、あらゆるプログラムが使うべき基本テクニック、構築の原則に重点を置く。

サンプルの選択は、コンパイラ、基本ライブラリ、シミュレーションという私の出身分野を反映している。サンプルは、現実のコードの単純化されたバージョンである。プログラミング言語と設計のポイントがディティールに埋もれないようにするために、単純化はどうしても必要である。しかし、対応する現実のコードを持たない、“キュート”なサンプルはない。x、y 変数、A、B 型、f()、g() 関数を使って言語の詳細を示す専門的なサンプルは、可能な限り付録 C に集めてある。

サンプルコードでは、識別子は次のようにプロポーショナルフォントで表記されている。

```
#include <iostream>

int main()
{
    std::cout << "Hello, new world!\n";
}
```

固定ピッチフォントのコードを見慣れたプログラマには、この表記スタイルは最初は“不自然”に見えるかもしれない。しかし、一般に、テキストの提示方法として、プロポーショナルフォントは固定ピッチフォントよりも優れていると考えられている。プロポーショナルフォントを使えば、非論理的な改行の数も減る。私が試してみたところでも、しばらくすると、ほとんどの人々がこのスタイルの方が読みやすいと感じるようになった。

C++言語とライブラリの機能は、マニュアルのようなドライな形ではなく、可能な限りそれらを使う文脈のなかで示すようにした。本書で取り上げた言語機能と説明の詳細は、C++を効果的に使うためには何が必要かということについての私の考え方を反映して

いる。Andrew Koenigと私自身の共著による姉妹書、“The Annotated C++ Language Standard”は、言語の完全な定義と定義を理解しやすくするためのコメントから構成されている。論理的には、“The Annotated C++ Standard Library”というもう1冊の姉妹書が必要なところだが、執筆のための時間と私の能力の限界から、そのような本を書くことを約束することはできない。

本書の他の部分は、§2.3.4（第2章第3節第4項）、§B.5.6（付録B5.6節）、§6.6[10]（第6章練習問題10）という形式で参照する。若干の強調箇所（たとえば、“文字列リテラルにはアクセス**できない**”）、重要な概念の初出箇所（たとえば、**多相化**）、C++文法の終端子（たとえば、*for-statement*）、コード例のコメントでは斜体を使っている†。コード例の識別子、キーワード、数値には太めの斜体字を使っている（たとえば、*class*、*counter*、*1712*）。

1.1.2 練習問題

各章の末尾には、練習問題が付けられている。ほとんどのものは、“～するプログラムを書きなさい”というタイプのものである。かならず、解答として、コンパイルできて少なくともいくつかのテストケースで正しく実行されるコードを書くようにしていただきたい。練習問題の難度はまちまちなので、その度合を推定してマークを付けてある。この度合は指数的なもので、（*1）に10分かかる場合なら、（*2）は1時間、（*3）は1日かかるだろう。プログラムを書き、テストするために必要な時間は、練習問題自体よりも読者の経験によって変わる。プログラムの実行のために新しいシステムに慣れなければならない場合には、（*1）の練習問題を解くために1日かかるかもしれない。しかし、手元にちょうどよいプログラムのコレクションを持っている読者なら、（*5）の練習問題でも、1時間で解ける場合がある。

Cプログラミングの本は、第1部の練習問題として使うことができるだろう。データ構造とアルゴリズムの本は、第2、第3部の練習問題として使うことができるはずである。

1.1.3 処理系についてのコメント

本書で使っている言語は、C++標準［C++, 1997］で定義された“純粋C++”である。そのため、練習問題はすべての処理系で動作するはずである。本書の主要なコードは、いくつかの処理系でテストされている。C++言語の一部としてつい最近採用されたばかりの機能を使っているサンプルは、一部の処理系ではコンパイルできなかった。しかし、どの処理系がそのサンプルをコンパイルできなかったかを示しても無意味である。実装者たちは、それぞれの処理系がC++のすべての機能を正しく受け付けられるようにするために懸命の作業を行っているので、その類の情報はすぐに古くなる。古いC++コンパイラやCコンパイラ向けに書かれたコードとの付き合い方については、付録Bをヒントにしていただきたい。

† 編集部注：日本語版においては、日本語の強調箇所には太字の明朝体を用い、コード例のコメントは立体としている。また、本文中ではコードの識別子、キーワードを太字の立体で示している。

1.2 C++の学習方法

C++言語を学習する上でもっとも大事なことは、概念に神経を注ぎ、言語の専門的なディティールに目を奪われないことである。プログラミング言語を学習するのは、よりよいプログラマになるためである。新システムの設計、実装、古いシステムの維持で、より力を発揮できるようになるためである。そのためには、詳細を理解することよりも、プログラミングと設計のテクニックを身につけることの方がはるかに重要だ。詳細の理解は、時間と経験とともに得られる。

C++は、さまざまなプログラミングスタイルをサポートする。これらのスタイルは、どれも静的な型チェックを基礎としており、大半のものは高いレベルでの抽象化とプログラマのアイディアの直接的な表現を目指している。それぞれのスタイルは、実行時の処理効率、空間の利用効率を維持しながら、それぞれの目標を充分に達成している。さまざまな言語（C、Fortran、Smalltalk、Lisp、ML、Ada、Eiffel、Pascal、Modula-2 など）の経験を持つプログラマでも、C++のメリットを引き出すためには、C++に適したプログラミングスタイルとテクニックを学習し、身につけるために時間を使わなければならないということを肝に銘ずる必要がある。

何も考えずに、ある言語で効果的なテクニックをほかの言語に当てはめてみても、ぎごちなく、効率が悪く、維持しにくいコードになるばかりである。そのようなコードは、書く身にとっても辛い。すべての行、すべてのコンパイラエラーが、“ 昔の言語 ”ではないことを思い出させるからである。Fortran、C、Smalltalk のスタイルで書いても構わないが、異なる思想のもとに作られた言語でそのようなことをしても、楽しくないし、経済的でもない。すべての言語は、C++プログラムの書き方のアイディアを豊富に供給してくれる源泉になり得るが、異なる文脈を持つ場所で効率を求めるなら、それらのアイディアは、C++の一般構造と型システムにフィットするものに変換する必要がある。言語の基本型システムを無視したところでは、ピュロスの勝利しか得られない。

C++は、段階的に学習できる言語である。新しいプログラミング言語を学習するためのアプローチは、すでに知っているのは何かということと、何を学習しようとしているのかということによって決まる。万人に向いたアプローチはない。私は、読者がよりよいプログラマ、設計者になるために C++を学習しているのだという前提で筆を進める。つまり、単に以前していたことをするための新しい構文を学習するためではなく、システム構築のための新しいよりよい方法を学習するために C++を身につけようとしているのだということである。新しく技能を身につけるためには時間がかかり、練習が必要なので、学習は段階的に進める必要がある。新しい自然言語や楽器を習得するためにかかる時間を考えてみていただきたい。よりよいシステムデザイナになるのは、これよりも簡単だし、時間もかからないが、ほとんどの人たちが望むほど簡単ではないし、時間もかかる。

おそらく、読者は言語のすべての機能とテクニックを理解する前に、C++を、それも実際のシステムの構築のために使うことになるだろう。C++は、複数のプログラミングパラダイムをサポートすることによって（第 2 章参照）、さまざまなレベルのプログラマの生産性を上げる。個々の新しいプログラミングスタイルは、読者の道具箱に新しいツールを追

加していくが、それらはそれ単独で効果を持ち、それ単独で読者のプログラマとしての能力を高める。C++は、基本的に直線的な形で概念を学習でき、学習過程のさまざまな段階で現実的なメリットが得られるように構成されている。これは、労力をつぎ込んだ度合にほぼ比例して効果が得られるということであり、非常に重要なポイントである。

C++の前にCを学習する必要があるかどうかについては、未だに議論が分かれるところだが、私はC++に直接行くのがベストだと確信している。CよりもC++の方が安全であり、表現力が高く、低水準のテクニックに神経を集中しなければならない状況が少なくなる。CとC++の共通サブセットを知り、C++が直接サポートしている高水準のテクニックの一部を知ったあとなら、Cが高水準機能を持たないために必要になってくるCの難しい部分も学習しやすくなるだろう。付録Bは、あえて言うなら古臭いコードを扱うためにC++からCに進むプログラマのためのガイドである。

C++には、独立して開発され、流通に回されている処理系が複数ある。ツール、ライブラリ、ソフトウェア開発環境も豊富に揃っている。C++の最新機能、使い方、ツール、ライブラリ、処理系などについての教科書、マニュアル、雑誌、ニューズレター、電子掲示板、メーリングリスト、コンファレンス、講習も無数にある。C++を本格的に使うつもりなら、それらの情報源を利用することをお勧めする。個々のものにはそれぞれの強調点とバイアスがあるので、最低でも2つ使うとよい。たとえば、［Barton, 1994］、［Booch, 1994］、［Henricson, 1997］、［Koenig, 1997］、［Martin, 1995］を検討してみていただきたい。

1.3 C++の設計

単純さは、C++の設計の重要な基準の1つだった。言語定義の単純さとコンパイラの単純さの二者択一を迫られたときには、かならず前者を選んだ。しかし、Cとの互換性の維持にも大きな重点が置かれていたので、Cの構文をすっきりさせることは最初から禁じ手になっていた。

C++には、組み込みの高水準型や高水準演算原素はない。たとえば、C++言語は、逆行列演算子を持つ行列型や結合演算子を持つ文字列型を提供しない。そのような型が必要なユーザーは、言語自体の枠組みのなかで型を定義できる。実際、新しい汎用型やアプリケーション固有型の定義は、C++プログラミングのもっとも基本的な作業である。うまく設計されたユーザー定義型と組み込みデータ型の唯一の違いは、使われ方ではなく、定義のし方である。第3部で取り上げるC++標準ライブラリには、そのような型と使い方の例が豊富に含まれている。ユーザーの視点から見れば、組み込みデータ型と標準ライブラリが提供する型の間には、ほとんど違いはない。

使わない場合でも実行時のオーバーヘッドやメモリのオーバーヘッドを引き起こすような機能は、C++では避けられている。たとえば、すべてのオブジェクトに“管理情報”を格納しなければならなくなるような言語要素を使うことは避けたので、2つの16ビット要素から構成される構造体を宣言した場合、その構造体は32ビットレジスタにセットできる。

C++は、伝統的なコンパイル、実行時環境、すなわちUNIX上のCプログラミング環

境で使うことを想定して設計されたが、幸い、C++はUNIXからの制限を受けなかった。UNIXとCは、単純に言語、ライブラリ、コンパイラ、リンカ、実行環境などの関係のモデルとして使われただけである。C++がほとんどすべてのプラットフォームで成功を収めたことについては、この小さなモデルの貢献が大きい。しかし、これよりもはるかに多くのサポートを提供する環境でも、C++には使い道がある。ダイナミックロード、差分的コンパイル、型定義データベースなどの機能は、言語に影響を与えることなく追加できる。

C++の型チェックとデータ隠蔽の機能は、プログラムが誤ってデータを破壊するのを防ぐコンパイル時の分析に依存している。意図的に原則を破る人間からの隠蔽や防御の機能はない。しかし、C++のデータ隠蔽、型チェック機能は、実行時のオーバーヘッドや空間的なオーバーヘッドを受けずに自由に使うことができる。役に立つ言語機能は、エレガントであるばかりでなく、現実のプログラムの文脈で手ごろに使えるものでなければならないという考え方である。

C++の設計についての系統的で詳細な説明については、［Stroustrup, 1994］を参照していただきたい。

1.3.1 効率と構造

C++は、Cプログラミング言語から開発されており、若干の例外を除き、Cをサブセットとしてサポートしている。基底言語であるC++のCサブセットは、コンピュータが直接操作するオブジェクト（数値、文字、アドレス）と型、演算子、文の間に密接な対応関係が生まれるように設計されている。**new**、**delete**、**typeid**、**dynamic_cast**、**throw**演算子と*try-block*を除けば、C++の個々の式、文は、実行時のサポートを必要としない。

C++は、Cと同じ、あるいはそれよりも効率的な関数呼び出し、リターンシーケンスを使うことができる。比較的効率的なこのようなメカニズムでも高くつく場合には、C++関数はインライン展開することもできる。実行時のオーバーヘッドを0にしつつ、便利な関数の記述形式を使うことができるのである。

Cのもともとの目標は、もっとも要求の厳しいシステムプログラミング分野の仕事でアセンブリ言語に取って代わることだった。C++を設計するときも、Cがこの分野で獲得した成果を失わないように充分な注意を払った。CとC++の違いは、主として型と構造の強調の度合である。Cは表現力が高く、制限が緩やかである。C++はC以上に高い表現力を持つ。しかし、表現力が増した分、オブジェクトの型にも今まで以上の注意を払う必要がある。オブジェクトの型がわかっていれば、それまでプログラマが苦痛なほど細かく動作を指定しなければならなかった式でも、コンパイラは正しく処理できる。オブジェクトの型がわかっていれば、それまでテスト時、あるいはさらにそのあとになるまで見つけられなかったようなエラーでも、コンパイラは適切に検出できる。型システムを使って関数の引数をチェックしたり、偶発的な破壊からデータを保護したり、新しい型を提供したり、新しい演算子を提供したりしても、C++では実行時や空間のオーバーヘッドが増えないことに注意していただきたい。

C++が構造を重視するのは、Cが設計されたとき以降のプログラムの規模の拡大を反映

している。1000 行ほどの小さなプログラムなら、優れたプログラミングスタイルのあらゆる原則を破っても、力づくで作ることもできるだろう。しかし、もっと大規模なプログラムでは、そのようなことは単純に不可能である。10 万行のプログラムでは、構造がしっかりしていなければ、古いエラーを 1 つ取り除くたびに新しいエラーが生まれるということになるだろう。C++は、大規模なプログラムに合理的な構造を与え、1 人の人間が従来よりもはるかに大量のコードを扱っても問題がないように設計されている。また、C++の設計目標には、C++の平均的な 1 行が、C や Pascal の平均的な 1 行よりもはるかに多くの内容を表現できるようにすることも含まれていた。現在の C++は、これらの目標を達成して余りあるほどになっている。

すべてのコードが、適切に構造化され、ハードウェアに依存せず、簡単に読めるということはあり得ない。C++は、安全性やわかりやすさを犠牲にせずに、ハードウェアの機能を直接的で効果的な方法で操作するための機能を持っている。C++は、これらのコードをエレガントで安全なインターフェイスの背後に隠蔽するための機能も持っている。

従来よりも大規模なプログラムで C++を使えば、当然、複数のプログラマがチームを組んで C++を使うことになる。モジュール化、強く型付けされたインターフェイス、柔軟性を強調する C++が威力を発揮するのは、このような場面である。C++は、他の言語と同程度に大規模プログラムを書くための機能を提供している。しかし、プログラムの規模が大きくなると、開発、維持に関わる問題は、言語の問題からツール、管理のより大きな問題にシフトしてくる。第 4 部では、これらの問題のいくつかを深く追究する。

本書では、一般に使える機能、一般に役に立つ型、ライブラリを提供するためのテクニックを強調していく。これらのテクニックは、大規模なプログラムを書くプログラマばかりでなく、小規模プログラムのプログラマにも役に立つだろう。また、一定以上の規模を持つプログラムは、互いに独立しつつ、少しずつの依存関係も持つ無数の部品から構成されるので、このような部品を書くためのテクニックは、あらゆるプログラマの役に立つだろう。

従来よりも木目の細かい型構造でプログラムを規定していくと、プログラムのソーステキストも大きくなってしまうのではないかという懸念があるかもしれない。C++には、そのような問題はない。関数の引数の型を宣言し、クラスを使う C++プログラムは、これらの機能を使わない同じ機能の C プログラムよりも、少し短くなることが多い。もちろん、機能する C バージョンを作ることができたらの話ではあるが、ライブラリを使えば、C++プログラムは C 版よりもはるかに短くなるだろう。

1.3.2 設計思想についてのコメント

プログラミング言語は、2 つの関連し合う目標を持つ。つまり、実行すべき動作を指定するための手段をプログラマに提供することと、何が実現可能かを考えるときの道具として一連の概念をプログラマに提供することである。第 1 の目標を達成するためには、一般に、“マシンに近い”言語が必要である。マシンのすべての重要な側面を、単純かつ効率的でプログラマが直観的に理解できる形で処理できなければならない。C 言語は、もともと

このことを念頭に置いて設計された。それに対し、第 2 の目標を達成するためには、解決方法のコンセプトを直接的、かつ簡潔に表現するために、“ 解決すべき問題に近い ”言語が必要である。C++を作るために C に追加された機能は、このことを念頭に置いて設計されている。

考え/プログラミングする言語とプログラマが想像できる問題、解決方法は、密接に結びついている。そのため、プログラマのエラーを減らす意図のもとに言語の機能に制限を加えるのは、良く言っても危険である。自然言語の場合と同様に、少なくとも 2 つ以上の言語を理解していることには、大きなメリットがある。言語は、プログラマに概念操作のツールを提供する。ある仕事に対して不充分な道具は、まず使われないだろう。エラーのない優れた設計は、特定の言語機能の有無だけで保証されるものではない。

型システムは、少しでも複雑な仕事では大きな威力を発揮する。実際、C++のクラス概念は、概念ツールとして強力なものであるということを実証してみせた。

1.4 歴史的背景

私は、C++を開発し、初期の言語定義を書き、最初の実装（処理系）を製作した。私は C++の設計基準を選択、定式化し、すべての主要機能を設計し、C++標準委員会に提出された言語拡張提案の処理責任者でもあった。

明らかに、C++は C［Kernighan, 1978］に多くを負っている。C はサブセットとして C++のなかに残されている。要求の厳しいシステムプログラミングの仕事に対応できるような低水準機能を重視する C の考え方も、C++に受け継がれている。C 自体は、その前身である BCPL［Richards, 1980］から多くの影響を受けている。実際、BCPL の//というコメントの記法は、C++に（再）導入された。C++のもう片方の生みの親は、Simula67［Dahl, 1970］［Dahl, 1972］である。C++の派生クラスと仮想関数を持つクラス概念は、Simula67 から借りたものである。また、C++の演算子多重定義機能と文を書けるあらゆる場所に宣言を配置できる自由は、Algol68［Woodward, 1974］とよく似ている。

本書の初版が発行されて以来、C++言語には大幅な見直しと改良が加えられた。大きな見直しの対象となったのは、多重定義の解決、リンク、メモリ管理機能といった分野である。また、C との互換性を高めるために、いくつかの小さな変更も加えられた。また、いくつかの一般化と大きな拡張機能も加えられた。それは、多重継承、**static** メンバ関数、**const** メンバ関数、**protected** メンバ、テンプレート、例外処理、実行時型識別、名前空間などである。これらの拡張、改訂を貫くテーマは、ライブラリを書き、使うための言語として C++をより良いものにすることだった。C++の発展の歴史は、［Stroustrup, 1994］に書かれている。

テンプレート機能は、主として静的に型付けされたコンテナ（リスト、ベクタ、マップなど）をサポートし、それらのコンテナをエレガントかつ効率的に利用できるようにするために設計された（ジェネリックプログラミング）。主要な目的は、マクロとキャスト（明示的な型変換）を削減することである。テンプレートは、Ada の generic（その強さも弱さも）と Clu のパラメータ化されたモジュールから部分的に触発されている。同様に、C++の例

外処理メカニズムは、Ada［Ichbiah, 1979］、Clu［Liskov, 1979］、ML［Wilstrom, 1987］から部分的に触発されている。1985年から1995年までのその他の新機能（多重継承、純粋仮想関数、名前空間）は、他の言語から取り込んだアイディアではなく、C++の利用経験から得られたものを一般化したものである。

集合的に"C with Classes"（クラスを持つC）［Stroustrup, 1994］と呼ばれていたC++の初期バージョンは、1980年から使われていた。この言語を開発したのは、効率面を除けばSimula67を使うのが理想的だったイベント駆動型のシミュレーションを書きたかったからである。"C with Classes"は、最小限の時間と空間でプログラムを書くことが厳しくテストされるような大規模プロジェクトで使われていた。この言語には、演算子の多重定義、リファレンス、仮想関数、テンプレート、例外処理などのディティールは含まれていなかった。研究所以外でC++が初めて使われたのは、1983年7月のことである。

C++という名前（"シープラスプラス"と発音される）は、1983年夏にRick Mascittiが考え出したものである。++はCのインクリメント演算子であり、この名前には、Cを発展させたものだという意味合いが込められている。"C+"という名前は構文エラーであり、まったく関係のない言語の名前としても使われている。Cのセマンティクスにうるさい人たちは、C++の方が++Cよりも劣っていると言っている。C++は、Cの拡張であり、機能の削減によってCの問題点を解決しようとはしていないので、Dとは呼ばれない。C++という名前のその他の解釈方法については、［Orwell, 1949］の付録を参照していただきたい。

C++は、私の友人たちと私がアセンブラ、C、その他のさまざまな高水準言語でプログラムを書かなくても済むようにすることを主目的として設計された。C++の目標は、個々のプログラマがより簡単に、かつ楽しい気持ちで優れたプログラムを書けるようにすることだった。初期のC++には紙の設計書はなかった。設計とドキュメンテーションと実装は同時進行していたのである。"C++プロジェクト"や"C++設計委員会"は存在しなかった。C++は、一貫して、ユーザーが感じた問題点に対処することを目的として、友人、同僚と私が交わした議論に基づいて発展してきた。

その後、C++は爆発的に成長し、状況が変わってきた。1987年のある時点でC++の正式な標準化が避けられないものであることが明らかになり、私たちは標準化の基礎を整える作業をスタートしなければならなかった［Stroustrup, 1994］。その結果、紙、電子メール、C++委員会を始めとするフェイスツーフェイスの会合を通じて、C++コンパイラの実装者、主要ユーザーとの間でコンスタントに連絡を取り合うという意図的な作業が始まった。

AT&T Bell Laboratoriesは、私がC++リファレンスマニュアルの改訂バージョンを実装者やユーザーと共有することを認めることによって、C++の標準化作業に大きく貢献した。これらの人々の多くは、AT&Tのライバルと見なされる企業のために働いているだけに、AT&Tの貢献の重要度は見逃すべきではない。もっと考え方の遅れた企業なら、何もしないことによって、言語がばらばらになるという重大な問題を引き起こしていただろう。しかし、現実には、数十の企業から約100名の人々が集まり、その後一般に承認されたリファレンスマニュアルになったものとANSI C++標準化作業のベースドキュメントを読み、

コメントを加えることができた。彼らの名前は、“ The Annotated C++ Reference Manual ”［Ellis, 1989］に書かれている。最終的に、ANSI X3J16 委員会は、Hewlett-Packard のイニシアティブのもとに、1989 年 12 月に招集された。1991 年 6 月には、この ANSI C++標準化作業（アメリカ国内）は、ISO の C++標準化作業（国際）の一部となった。1990 年以来、この合同 C++標準委員会は、C++の発展と言語定義の改訂の主要な討論の場となった。私は、一貫してこれらの委員会に参加した。特に、私は言語拡張の作業グループの議長として、大きな変更、新言語機能追加の各種提案を処理する責任者として働いた。一般の評価のための最初の標準案は、1995 年 4 月に作られた。C++国際標準の正式な承認は、1998 年に予定されている。

C++は、本書で示すいくつかのクラスと手を携えて発展してきた。たとえば、私は、演算子多重定義のメカニズムとともに複素数、ベクタ、スタッククラスを開発した。文字列、リストクラスも、同じ作業の一貫として、Jonathan Shopiro と私によって開発された。Jonathan の文字列、リストクラスは、ライブラリの一部として多用された最初のクラスである。標準 C++ライブラリの文字列クラスは、これらの初期の作業に起源を持っている。［Stroustrup, 1987］と §12.7 に書かれているタスクライブラリは、初めて書かれた“ C with Classes ”プログラムの一部だった。私は、Simula スタイルのシミュレーションをサポートするためにこれと関連クラスを書いた。タスクライブラリは、主として Jonathan Shopiro によって改訂、再実装されており、現在でも多用されている。本書第 1 版に書かれているストリームクラスは、私が設計、実装したものである。Jerry Schwarz は、Andrew Koenig のマニピュレータテクニック（§21.4.6）などのアイディアを使って、これを iostreams ライブラリ（第 21 章）に変身させた。iostreams ライブラリは、標準化の過程で主として Jerry Schwarz、Nathan Myers、Norihiro Kumagai の手によってさらに磨きをかけられた。テンプレート機能の開発は、Andrew Koenig、Alex Stepanov、私などによって作られた **vector**、**map**、**list**、**sort** テンプレートの影響を受けた。さらに、テンプレートを使ったジェネリックプログラミングについての Alex Stepanov の仕事は、標準 C++ライブラリのコンテナ、アルゴリズムの部分を導いた（§16.3、第 17 章、第 18 章、§19.2）。数値演算のための **valarray** ライブラリ（第 22 章）は、主として Kent Budge の仕事である。

1.5 C++の用途

C++は、基本的にあらゆる応用分野で、数十万人のプログラマによって使われている。C++がこれだけ普及しているのは、約 10 種類ほどの独立した処理系、数百のライブラリ、数百の教科書、数種類の雑誌、数多くのコンファレンス、無数のコンサルタントに支えられているからである。さまざまなレベルでの教育訓練コースも広く普及している。

初期の応用分野は、システムプログラミングに大きく偏っていた。たとえば、いくつかの大きなオペレーティングシステムが C++で書かれており（［Campbell, 1987］［Rozier, 1988］［Hamilton, 1993］［Berg, 1995］［Parrington, 1995］）、主要部が C++で書かれているオペレーティングシステムはほかにもたくさんある。私は、低水準処理に対する妥協のない効率の追求は、C++にとって本質的に重要だと考えていた。そのため、C++は、リ

アルタイムの制約のもとでハードウェアの直接操作に依存するデバイスドライバなどのソフトウェアを書ける言語になった。このようなコードでは、パフォーマンスの予測可能性は、少なくとも生のスピードと同程度の重要性を持つ。作られたシステムがコンパクトになるかどうかも同様の重要性を持つことが多い。C++は、時間と空間について厳しい制約を課せられたコードですべての言語機能を利用できるように設計されている。

ほとんどのアプリケーションは、許容できるスピードで動かすために決定的に重要な部分を含んでいるものだが、大半のコードは、そのような部分ではない。それらのコードでは、維持、拡張、テストのしやすさが重要な意味を持つ。C++は、これらの問題をサポートしているので、信頼性が絶対必要条件である分野や、時間とともに要求が大きく変化する分野でも普及した。たとえば、銀行、商事、保険、電気通信、軍用アプリケーションである。アメリカの長距離電話システムの中央制御は、数年前からC++に依存しているし、800番（受信者負担の電話）は、C++プログラムによってルーティングされている［Kamath, 1993］。これらのアプリケーションの多くは、大規模である上に長寿である。そのため、C++の開発では、安定性、互換性、スケーラビリティが常に考慮された。数百万行のC++プログラムは、決して珍しいものではない。

C++は、Cと同様に数値演算を念頭に置いて設計された言語ではない。しかし、C++は数値演算、科学技術計算で多用されている。その大きな理由は、従来数値演算処理とされてきたものの多くが、グラフィックスや、伝統的なFortranの鋳型にはまらないデータ構造に依存する処理［Budge, 1992］［Barton, 1994］と結び付けて実行しなければならなくなっていることにある。グラフィックスとユーザーインターフェイスは、C++が非常に多用されている分野である。Apple MacintoshやWindows PCを使ったことがあれば、その人はC++を間接的に使ったことになる。これらのシステムの主要なユーザーインターフェイスは、C++プログラムなのである。さらに、X for UNIXをサポートするライブラリのなかでもっともポピュラーなもののいくつかも、C++で書かれている。つまり、C++は、ユーザーインターフェイスが重要な部分を占めるさまざまなアプリケーションで一般的に使われているということである。

C++の最大の長所は、以上からも明らかだろう。C++は、さまざまな応用分野を横断して動作しなければならないアプリケーションで効率よく使える力を持っているということである。LAN及びWAN、数値演算、グラフィックス、ユーザーとのやり取り、データベースアクセスをすべて必要とするアプリケーションは、非常に多い。従来、これらの応用分野は別個のものと考えられ、さまざまなプログラミング言語を使う別々の専門家集団によって担われてきた。しかし、C++は、これらすべての分野で広く使われるようになってきている。しかも、C++は他の言語で書かれたコード片やプログラムとも共存できる。

C++は、教育、研究用途でも広く使われている。このことは、C++は今まででもっとも小さく、もっともクリーンな言語ではないと（正しくも）指摘した幾人かを驚かせた。しかし、C++には、以下の特長がある。

- 基本概念をうまく教えられるだけのクリーンさを持つ
- 要求の厳しいプロジェクトに耐えられるだけの現実性、効率、柔軟性を持つ
- ばらばらな開発、実行環境に依存する組織、共同研究チームでも利用できる

- 高度な概念、テクニックを教育するための手段として充分包括的な内容を持っている
- 学習したことを非学術的な用途に転化させられるだけの商業性を持つ

C++は、ユーザーを育てる言語である。

1.6 CとC++

Cが C++の基底言語として選ばれた理由は、次のようなものである。

[1] Cは柔軟、簡潔で、比較的低水準である
[2] Cはシステムプログラミングのほとんどの仕事で充分使える
[3] Cはあらゆるところであらゆるマシンの上で実行されている
[4] Cは UNIX プログラミング環境にフィットしている

Cにはいくつかの問題があるが、0から設計された言語にも問題はあるだろう。そして、私たちはCの問題を知っている。Cを使ったからこそ、Cに Simula 風のクラスを追加したいという最初のアイディアから数ヶ月のうちに" C with Classes "が役に立つ（ぎごちないにしても）ツールになったということは重要なポイントである。

C++が以前よりも広く使われるようになり、その機能がCの機能を大きく凌駕するようになって、互換性を維持すべきかどうかという疑問は繰り返し提出されるようになった。Cから継承したいくつかのものを取り除けば、いくつかの問題は明らかに回避できる（たとえば、[Sethi, 1981] を参照）。しかし、このような道が選択されなかったのは、次の理由による。

[1] Cから C++への完全な書き直しが不要であれば、C++によって生き返るCコードが無数にある。
[2] C++がCに対するリンク互換性を持ち、構文的に非常に近ければ、C++プログラムでも利用できるCで書かれたライブラリ関数、ユーティリティソフトウェアコードが無数にある。
[3] Cを知っており、基礎から学び直さなくても、C++の新機能を学ぶだけで C++を使えるプログラマが数十万人もいる。
[4] C++とCは、同じ人間によって同じシステムで何年も前から使われているので、ミスや混乱を最小限に抑えるためには、両者の違いは非常に大きいか非常に小さいものでなければならない。

C++の定義は、Cでも C++でも認められている言語要素がどちらの言語でも同じ意味を持つように改訂されてきている（§B.2）。

C言語自体にも、C++の開発の影響を受けて進化している部分がある [Rosler, 1984]。ANSI C 標準 [C, 1990] には、" C with Classes "から借りた関数宣言構文が含まれている。成果の借用は、双方向的である。たとえば、ポインタ型の **void***は、最初は ANSI C のために考えられたものだが、最初に実装されたのは C++である。本書の第1版で約束したように、不要な非互換性を取り除くために、C++の定義は見直されている。今の C++は、

以前よりもCに対する互換性が高い。理想は、C++をできる限りANSI Cに近付け、それ以上は近付けないことだった［Koenig, 1989］。100%の互換性は、型の安全性やユーザー定義型と組み込み型のスムースな統合に妥協を強いることになるので、一度も目標になったことはない。

C++を学習するためにCを知っている必要はない。Cプログラミングは、C++によって不要になったさまざまなテクニックやトリックを助長する。たとえば、C++では、Cと比べて明示的な型変換（キャスト）が必要になる頻度は低い（§1.6.1）。しかし、**優れた**Cプログラムは、C++プログラムにもなるものである。たとえば、Kernighan and Ritchie, " The C Programming Language 2nd Edition, "［Kernighan, 1988］に含まれているすべてのプログラムは、C++プログラムである。C++を学習するときには、静的に型付けされた言語の経験は役に立つ。

1.6.1 Cプログラマのためのヒント

Cを知っていればいるほど、CスタイルでC++を書き、C++の潜在的なメリットを失うのは避けられないことのようである。CとC++の違いを説明した付録Bをぜひ見ていただきたい。CよりもC++の方がすぐれた方法を持っている場面について、簡単に説明しておこう。

[1] C++では、マクロはほとんどまったく不要である。名前付き定数を定義するときには**const**（§5.4）か**enum**（§4.8）を使い、関数呼び出しのオーバーヘッドを避けたいときには**inline**（§7.1.1）、関数や型のファミリを指定したいときには**template**（第13章）、名前の衝突を避けたいときには**namespace**（§8.2）を使う。

[2] 変数はすぐに初期設定できるので、必要になる前に宣言しない。宣言は、文を配置できるあらゆる場所（§6.3.1）、*for-statement*（§6.3.3）、条件部（§6.3.2.1）に配置できる。

[3] mallocを使わない。**new 演算子**（§6.2.6）は、同じ仕事をよりよくこなす。また、**realloc()**の代わりに、**vector**（§3.8）を試してみていただきたい。

[4] 関数やクラスの実装の奥底を除き、**void***、ポインタ演算、共用体、キャストを避けるようにする。ほとんどの場合、キャストは設計エラーの兆候である。明示的な型変換を使わなければならないときには、しようとしていることをより正確に記述する" 新しいキャスト "（§6.2.7）を使うようにする。

[5] 配列やCスタイルの文字列を使う場所を最小限に抑える。C++標準ライブラリの**string**（§3.5）、**vector**（§3.7.1）は、伝統的なCのスタイルよりもプログラミングを単純化できる。一般に、標準ライブラリがすでに提供しているものを自分で作ろうとはしない方がよい。

CのリンクAS方式に従うC++関数を作るためには、Cリンクを持つことを宣言しなければならない（§9.2.4）。

特に重要なのは、一連の構造体があり、関数がそれらのビットを操作していると考えるのではなく、クラスとオブジェクトが表現するやり取りの形としてプログラムを考えることである。

1.6.2 C++プログラマのためのヒント

すでに多くの人々がC++を使って10年になる。単一の環境でC++を使っている人々はそれ以上であり、それらの人々は、初期のコンパイラや第1世代のライブラリが持っていた制限に自分を合わせて何とか仕事をこなしていくことを学んできている。ベテランのC++プログラマが気付かずにいることは、新しい決定的に重要なプログラミングテクニックを実現可能にしたものが、このような新機能の導入ではなく、機能の間の関係の変化だということである。つまり、初めてC++を学んだときに役に立たないと思ったり感じたりしたものが、現在では優れたアプローチになっているかもしれないということである。このことは、基礎を再点検する以外に理解できない。

本書を章の順番どおりに読んでいただきたい。章の内容をすでに知っている場合には、数分で読み通せるだろう。内容を知らない章では、何か予期せぬことを学習するはずである。私自身、本書を書くことによってかなりのことを学んだ。そして、本書に書いた機能やテクニックを知っているC++プログラマはほとんどいないのではないかと思った。言語を活用するためには、一連の機能やテクニックに秩序をもたらす展望が必要である。本書は、構成とサンプルを通じてそのような展望を提供するはずである。

1.7 C++プログラミングについての考え方

プログラムの設計の仕事は3段階で進められれば理想的である。まず、問題の明確な理解を獲得し（分析）、解決方法を構成する主要なコンセプトを明らかにし（設計）、最後にその解決方法をプログラムのなかに表現する（プログラミング）。しかし、問題の詳細や解決方法のコンセプトは、それらをプログラムのなかに表現し、まずまずの成果を得ようとする作業のなかでしか明確に把握できないことが多い。そこで、プログラミング言語の選択が大きな意味を持つことになる。

ほとんどのアプリケーションには、基本データ型のどれかや、データとの結びつきを持たない関数では簡単に表現できないコンセプトが含まれている。そのようなコンセプトがあれば、プログラムのなかでそれを表現するクラスを宣言する。C++のクラスは、型である。つまり、C++クラスは、そのクラスのオブジェクトがどのように行動するかを規定する。どのように作成し、どのように操作でき、どのように解体されるか。クラスは、オブジェクトがどのように表現されるかも規定することがあるが、それは設計の初期の段階では、大きな問題ではない。優れたプログラムを書くための鍵は、個々のクラスが1つのコンセプトを明確に表現できるようにクラスを設計することである。このことは、次のような疑問に神経を集中させなければならないということである場合が多い。すなわち、このクラスのオブジェクトはどのようにして作成されるか？　このクラスのオブジェクトはコピーしたり解体したりすることができるか？　そのようなオブジェクトに加えられる演算は何か？　これらの疑問にはっきりと答えられない場合、そのコンセプトは最初は“クリーン”ではなかったということだろう。その場合、すぐに問題をめぐってコードを書くよりも、問題と解答案についてもっとよく考えた方がよいかもしれない。

一番簡単に扱えるコンセプトは、数値、集合、幾何学図形など、伝統的な数学的形式を持つコンセプトである。標準C++ライブラリには、テキスト入出力、文字列、基本コンテナ、コンテナ操作の基本アルゴリズム、いくつかの数学的クラスが含まれている（第3章、§16.1.2）。また、一般的なコンセプトや特定の領域に固有なコンセプトを扱う驚くほどバラエティに富んだライブラリが用意されている。

1つのコンセプトは、それだけで存在するわけではない。かならず、関連するコンセプトとともに房のような塊を作っている。プログラムのなかにクラスの関係を組織していくこと、すなわち解答のなかに含まれているさまざまなコンセプトの間の正確な関係を判定していくことは、最初の個々のクラスを並べる作業よりも難しいことが多い。個々のクラス（コンセプト）が他のすべてのクラスに依存するような混乱した関係は望ましくない。A、Bの2つのクラスがあったとする。"AがBのデータを使う"というような関係は一般に削減できるが、"AがBの関数を呼び出す"、"AがBを作成する"、"AがBメンバを持つ"といった関係が大きな問題の原因になることはまずない。

階層的な秩序を与えることは、複雑さを管理するためのもっとも強力な知的ツールの1つである。一番一般的なコンセプトを根元に置いて、関連するコンセプトを木構造にまとめるのである。C++では、派生クラスがこのような構造を表現している。プログラムは、一連の木構造、すなわちクラスの非循環有向グラフとして構成されることが多い。つまり、プログラマは、それぞれ派生クラスを持ついくつかの基底クラスを規定するのである。仮想関数（§2.5.5、§12.2.6）は、コンセプトのもっとも一般的なバージョン（基底クラス）に対する演算を定義するために使えることが多い。必要なら、仮想関数の解釈は特定のケース（派生クラス）に合わせて調整することができる。

クラスの非循環有向グラフでも、プログラムのコンセプトを組織するためには不充分に見えることがある。一部のコンセプトは、本質的に相互依存するものらしい。そのような場合には、循環的な依存関係を局所化してプログラムの全体構造に影響を与えないようにする。そのような相互依存性を削減、局所化できない場合には、どのようなプログラミング言語にも助けてもらえないような窮地に立たされることになるだろう。基本コンセプトの間に簡単に表現できる関係を結べない場合、プログラムは管理不能になる可能性が高い。

もつれた依存グラフをほどく最良のツールの1つは、インターフェイスとインプリメンテーション（実装）の明確な分離である。抽象クラス（§2.5.4、§12.3）は、この目的のためにC++が提供するメインツールである。

共通性には、テンプレートによって表現できる形のものもある（§2.7、第13章）。クラステンプレートは、クラスのファミリを規定する。たとえば、リストテンプレートは、"**T**のリスト"を規定する。この**T**はどのような型でもよい。つまり、テンプレートは、引数としてほかの型を取り、新しい型を生成するための方法を規定するメカニズムなのである。もっとも一般的なテンプレートは、リスト、配列、連想配列などのコンテナクラスとそれらのコンテナを使った基本アルゴリズムである。クラスと関連関数のパラメータ化を継承で表現するのは、一般に間違っている。このような場合には、テンプレートがもっとも適している。

プログラミングの大半は、基本データ型、構造体、プレーンな関数といくつかのライブ

ラリクラスだけで単純かつ明解に進められることを覚えておいていただきたい。新しい型を定義するためのメカニズムは、本当に必要な場合以外、使うべきではない。

“C++で優れたプログラムを書くためにはどうしたらよいか”という問は、“英語で優れた散文を書くためにはどうしたらよいか”という問とよく似ている。答は2つある。“自分が言いたいことをはっきりさせなさい”と“練習して、優れた作品を真似なさい”である。どちらの答も、英語と同じように C++にも当てはまるように感じられる。そして、守るのが難しい点でも似ている。

1.8 アドバイス

ここでは、C++を学習するときに頭に入れておいた方がよい“原則”をまとめておいた。熟練してきたら、読者のアプリケーションのタイプやプログラミングスタイルに適したものに発展させていけばよい。原則の文章はわざと非常に単純にしてあるので、ディティールはない。あまり文字面に縛られないようにしていただきたい。優れたプログラムを書くためには、知性と感性と忍耐が必要である。最初からうまくいくということはない。実際にやってみることだ!

[1] プログラムを書くときには、問題に対する解決方法に含まれる観念に具体的な表現を与えよ。それらの観念をできるだけ直接的にプログラムの構造に反映させるようにせよ。

[a]“それ”が独立した観念と考えられるなら、それをクラスにせよ。
[b]“それ”が独立した存在と考えられるなら、それをオブジェクトにせよ。
[c] 2つのクラスが共通インターフェイスを持つなら、そのインターフェイスを抽象基底クラスにせよ。
[d] 2つのクラスの実装のなかに目立つ共通点がある場合には、その共通点を基底クラスにせよ。
[e] あるクラスがオブジェクトのコンテナになっている場合には、それをテンプレートにせよ。
[f] 関数がコンテナに対するアルゴリズムを実装するものなら、コンテナファミリのアルゴリズムを実装するテンプレート関数にせよ。
[g] 一連のクラス、テンプレートなどが論理的に関連し合っているなら、それらを共通の名前空間に配置せよ。

[2] 行列や複素数といった数学的な存在ではないものを実装するクラスや、リストのような低水準型を定義するときには、以下の原則に従え。

[a] 大域データを使うな(メンバを使う)。
[b] 大域関数を使うな。
[c] 公開データメンバを使うな。
[d] [a]、[c] を避ける以外の目的でフレンドを使うな。

[e] クラス内に“ 型フィールド ”を作らず、仮想関数を使え。
[f] 最適化効果が大きい場合を除き、インライン関数を使うな。

各章の“ アドバイス ”の節には、より細かい個別的な経験則をまとめてある。このアドバイスは、あくまでもラフな経験則であり、不可侵の法則というわけではないことを忘れないでいただきたい。アドバイスは、“ 合理的な場面 ”のみに当てはまる。知性、経験、常識、感性に優るものはない。

私は、“ こうしてはならない ”という原則は、あまり役に立たないと思っているので、ほとんどのアドバイスは何をすべきかという形のヒントになっている。否定的な原則を示す場合には、絶対禁止事項に見えないようにしている。私は、C++の大きな機能のなかに適切な使い方のないものは存在しないことを知っている。“ アドバイス ”節には、説明はないが、本書の適切な部分の参照情報が付けられている。否定的なアドバイスが書かれているときには、参照箇所には代わりに使うべき方法が書かれていることが多い。

1.8.1 参考文献

本文には、直接的な参照はほとんど含まれていないが、直接、間接的に言及した書籍、論文を簡単にまとめておく。

[Barton,1994]　John J. Barton and Lee R. Nackman: *Scientific and Engineering C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 1-201-53393-6.

[Berg,1995]　William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38 No. 10. October 1995.

[Booch,1994]　Grady Booch: *Object-Oriented Analysis and Design*. Benjamin/Cummings. Menlo Park, Calif. 1994. ISBN 0-8053-5340-2.

[Budge,1992]　Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990]　X3 Secretariat: *Standard — The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.

[C++,1997]　X3 Secretariat: *Draft Standard — The C++ Language*. X3J16/97-14882. Information Technology Council (NSITC). Washington, DC, USA.

[Campbell,1987]　Roy Campbell, et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Coplien,1995]　James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. Reading, Mass. 1995. ISBN 1-201-60734-4.

[Dahl,1970]　0-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl,1972]　0-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction in Structured Programming*. Academic Press, New York. 1972.

[Ellis,1989]　Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Gamma,1995]　Eric Gamma, et al.: *Design Patterns*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-63361-2.

[Goldberg,1983] A. Goldberg and D. Robson: *SMALLTALK-80 — The Language and Its Implementation*. Addison-Wesley. Reading, Mass. 1983.

[Griswold,1970] R. E. Griswold, et al.: *The Snobol4 Programming Language*. Prentice-Hall. Englewood Cliffs, New Jersey. 1970.

[Hamilton,1993] G. Hamilton and P. Kougiouris: *The Spring Nucleus: A Microkernel for Objects*. Proc. 1993 Summer USENIX Conference. USENIX.

[Griswold,1983] R. E. Griswold and M. T. Griswold: *The ICON Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey. 1983.

[Henricson,1997] Mats Henricson and Erik Nyquist: *Industrial Strength C++: Rules and Recommendations*. Prentice-Hall. Englewood Cliffs, New Jersey. 1997. ISBN 0-13-120965-5.

[Ichbiah,1979] Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14 No. 6. June 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72 No. 5. September/October 1993.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice-Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language (Second Edition)*. Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible — but no closer*. The C++ Report. Vol. 1 No. 7. July 1989.

[Koenig,1997] Andrew Koenig and Barbara Moo: *Ruminations on C++*. Addison Wesley Longrnan. Reading, Mass. 1997. ISBN 1-201-42339-1.

[Knuth,1968] Donald Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Mass.

[Liskov,1979] Barbara Liskov et al.: *Clu Reference Manual*. MIT/LCS/TR-225. MIT Cambridge. Mass. 1979.

[Martin,1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.

[Orwell,1949] George Orwell: *1984*. Seeker and Warburg. London. 1949.

[Parrington,1995] Graham Parrington et al.: *The Design and Implementation of Arjuna*. Computer Systems. Vol. 8 No. 3. Summer 1995.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL — The Language and Its Compiler*. Cambridge University Press, Cambridge. England. 1980. ISBN 0-521-21965-5.

[Rosier,1984] L. Rosier: *The Evolution of C — Past and Future*. AT&T Bell Laboratories Technical Journal. Vol. 63 No. 8. Part 2. October 1984.

[Rozier,1988] M. Rozier, et a1.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1 No. 4. Fall 1988.

[Sethi,1981] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations*. Software Practice &, Experience. Vol. 11. 1981.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34(R. 1). August, 1994.

[Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Mass. 1986. ISBN 0-201-12078-X.

［Stroustrup,1987］ Bjarne Stroustrup and Jonathan Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ conference. Santa Fe, New Mexico. November 1987.

［Stroustrup,1991］ Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.

［Stroustrup,1994］ Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

［Tarjan,1983］ Robert E. Tarjan: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, Penn. 1983. ISBN 0-898-71187-8.

［Unicode,1996］ The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley Developers Press. Reading, Mass. 1996. ISBN 0-201-48345-9.

［UNIX,1985］ *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

［Wilson,1996］ Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge. Mass. 1996. ISBN 0-262-73118-5.

［Wikström,1987］ Åke Wikström: *Functional Programming Using ML*. Prentice-Hall. Englewood Cliffs, New Jersey. 1987.

［Woodward,1974］ P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office. London. England. 1974.

大規模ソフトウェアの設計、開発に関連した参考文献は、第 23 章の末尾にまとめてある。

第 2 章

C++ひとめぐり

手始めに言語の法律家を全員処分しよう
——*Henry VI, part II*

C++とは何か◆プログラミングのパラダイム◆手続き型プログラミング◆モジュラプログラミング◆分割コンパイル◆例外処理◆データ抽象◆ユーザー定義型◆具象型◆抽象型◆仮想関数◆オブジェクト指向プログラミング◆ジェネリックプログラミング◆コンテナ◆汎用アルゴリズム◆言語とプログラミング◆アドバイス

2.1 C++とは何か

C++は、システムプログラミング向きの傾向を持つ汎用プログラミング言語で、次の特長を持つ。

- C よりも優れている C
- データ抽象: data abstraction をサポートする
- オブジェクト指向プログラミング: object-oriented programming をサポートする
- ジェネリックプログラミング: generic programming をサポートする

本章では、言語定義の詳細に立ち入らずにこれらがどういう意味を持っているかを説明する。本章の目的は、C++の全般的な見取り図と C++を使うための主要テクニックを示すことであり、C++プログラミングをスタートさせるために必要な詳しい情報を提供すること**ではない**。

本章のなかにわかりにくいところがあれば、そこは無視して先に進んでいただきたい。すべてのことは、あとの章で詳しく説明される。しかし、本章の一部を読み飛ばした場合

には、あとで自分のために読み返すようにしていただきたい。

言語の機能を細かく知っているからといって（たとえ、それが言語の**すべての**機能だったとしても）、言語の全般的な見取り図や言語を使うための基本テクニックがなければ役には立たないのである。

2.2 プログラミングのパラダイム

オブジェクト指向プログラミングは、プログラミングのテクニックであり、一連の問題を解決する“優れた”プログラムを書くためのパラダイムである。“オブジェクト指向プログラミング言語”という言葉が意味を持つとしたら、それは、オブジェクト指向スタイルのプログラミングをよくサポートするメカニズムを持っているプログラミング言語ということでなければならない。

ここには、重要な区別がある。ある言語があるプログラミングスタイルを**サポートする**と言う場合、その言語はそのスタイルを使うために便利な（妥当なレベルに達したやさしさ、安全性、効率を持つ）機能を提供していなければならない。そのようなスタイルのプログラムを書くためにとてつもない労力や技能を必要とするような言語は、そのテクニックをサポートするとは言えない。単純にそのテクニックを使うことを可能にしているだけである。たとえば、Fortran77で構造化プログラムを書いたり、Cでオブジェクト指向プログラムを書いたりすることは不可能ではないが、これらの言語は対象のテクニックをサポートしていないので、仕事は不必要に難しくなる。

パラダイムのサポートということは、パラダイムを直接利用できる言語機能があるかどうかという明解な側面ばかりでなく、パラダイムからの意図的ではない逸脱をコンパイル時あるいは実行時あるいはその両方でチェックしているかどうかという目に見えにくい側面も含んでいる。型のチェックは、このようなサポートのなかではもっともわかりやすいものである。パラダイムに対する言語のサポートを拡張する手段としては、そのほかにあいまいさの検出や実行時チェックなどが使われる。ライブラリやプログラミング環境といった言語外の機能も、パラダイムサポートを強化できる。

ある言語が他の言語の持たない機能を持っているからといって、その言語がもう一方の言語よりも優れているということはない。逆の例も無数にある。重要なのは、言語が持つ機能ではなく、言語の持つ機能が所期の応用分野で所期のプログラミングスタイルをサポートするために充分かどうかである。

[1] すべての機能は、言語のなかにエレガントかつクリーンに統合されていなければならない。

[2] そのままでは余分な別個の機能が必要になるような解決方法が、機能の組み合わせによって実現できなければならない。

[3] 傍系的な“特殊目的”の機能は、できる限り減らさなければならない。

[4] ある機能の実装によってそれを必要としないプログラムに過大なオーバーヘッドを与えてはならない。

[5] 言語のうち、プログラムを書くために使われる機能だけしか知らなくても、プログラムが書けるようでなければならない。

第 1 の原則は、美観と論理性の問題である。次の 2 つは、ミニマリズムの理想を表現したものであり、最後の 2 つは、“ 知らない機能がプログラマを傷付けてはならない ”と要約できるだろう。

C++は、従来の C プログラミングテクニックに加え、C では制約されているデータ抽象、オブジェクト指向プログラミング、ジェネリックプログラミングをサポートするために設計された。すべてのユーザーに対して特定のプログラミングスタイルを強制することは**意図していない**。

以下の節では、いくつかのプログラミングスタイルと、それらをサポートする C++言語の主要なメカニズムについて考える。まず、手続き型プログラミングからスタートし、クラス階層を使うオブジェクト指向プログラミングとテンプレートを使うジェネリックプログラミングに向かって進む。個々のパラダイムは、その前身を基礎として構築されており、C++プログラマの道具箱に新しい道具を追加してくれる。また、それぞれのパラダイムは、実証済みの設計アプローチを反映したものになっている。

ここで示される言語機能は、網羅的なものではない。ここでは、言語の詳細ではなく、設計アプローチとプログラムの構成方法に重点を置く。この段階では、C++である問題を解決する方法を理解することよりも、C++でどのようなことができるかというイメージを掴むことの方がはるかに重要である。

2.3 手続き型プログラミング

最初のプログラミングパラダイムは、次のようなものである。

> 必要な手続きを決めよ。
> 見つけられる限りで最良のアルゴリズムを使え。

重点は処理、すなわち所期の演算を実行するために必要なアルゴリズムに置かれている。言語は、関数に引数を渡し、関数から戻り値を受け取るためのメカニズムを機能することによって、このパラダイムをサポートする。この思考方法で問題となるテーマは、引数の渡し方、異なる種類の引数や関数（手続き、ルーチン、マクロなど）の見分け方などである。

“ 優れたスタイル ”の典型的な例は、平方根関数である。この関数は、倍精度浮動小数点数の引数を取り、その平方根を返す。平方根関数は、目的を達成するために、よく知られた数学的な計算を実行する。

```
double sqrt(double arg)
{
    // 平方根を計算するためのコード
}

void f()
```

```
{
    double root2 = sqrt(2);
    // …
}
```

中括弧 {} は、C++におけるグループ分けを表現する。ここでは、関数本体の先頭と末尾を示している。ダブルスラッシュ//は、行末まで続くコメントの先頭を示す。キーワード **void** は、関数が値を返さないことを示す。

プログラムの構成という観点から見ると、関数は、アルゴリズムの迷路のなかに秩序を生み出す。アルゴリズム自体は、関数呼び出しやその他の言語機能を使って記述される。以下の項では、演算を表現するための C++の基本機能を簡単にスケッチする。

2.3.1 変数と算術演算

すべての名前とすべての式は、それに対して加えられる演算を決める型を持っている。たとえば、次の宣言は、**inch** が **int** 型であることを規定する。

```
int inch;
```

つまり、**inch** は、整数変数である。

宣言: *declaration* は、プログラムに名前を導入する文である。宣言は、名前の型を指定する。**型**: *type* は、名前や式の正しい使い方を決める。

C++はさまざまな基本データ型を提供しているが、それらはハードウェアの機能に直接対応している。たとえば、次のとおり。

```
bool        // 論理型。有効な値は true と false のみ
char        // 文字型。たとえば、'a'、'z'、'9'
int         // 整数型。たとえば、1、42、1216
double      // 倍精度浮動小数点数型。たとえば、3.14、299793.0
```

char 変数は指定されたマシンで文字を保持するために自然なサイズ（一般に1バイト）、int 変数は指定されたマシンで整数演算を行うために自然なサイズ（一般に1ワード）を持つ。

算術演算子は、これらの型の自由な組み合わせに対して使うことができる。

```
+           // 加算。単項と 2 項の両方
-           // 減算。単項と 2 項の両方
*           // 乗算
/           // 除算
%           // 剰余
```

比較演算子も、同様である。

```
==          // 等しい
```

```
!=          // 等しくない
<           // より小さい
>           // より大きい
<=          // 以下
>=          // 以上
```

代入、算術演算では、C++は基本データ型を適切に変換するので、これらの型を自由に織り交ぜて使うことができる。

```
void some_function()      // 値を返さない関数
{
    double d = 2.2;       // 浮動小数点数を初期設定する
    int i = 7;            // 整数を初期設定する
    d = d + i;            // 合計をdに代入する
    i = d * i;            // 積をiに代入する
}
```

C の場合と同様に、=は代入演算子であるのに対し、==は等しいかどうかをテストする。

2.3.2 テストとループ

C++には、選択とループを表現する伝統的な文が用意されている。たとえば、次に示すのは、ユーザーに対してプロンプトを表示し、ユーザーの応答を示す bool 値を返す単純な関数である。

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";   // 質問の書き込み

    char answer = 0;
    cin >> answer;                                   // 回答の読み出し

    if (answer == 'y') return true;
    return false;
}
```

<<演算子（“ ～へ出力 ”）は出力演算子として使われている。cout は標準出力ストリームである。>>演算子（“ ～から入力 ”）は入力演算子として使われている。cin は標準入力ストリームである。>>の右辺値は、どのような入力を受け付けるか、また入力処理のターゲットは何かを決める。出力文字列の末尾の \n 文字は、改行を表す。

このサンプルは、'n' という回答を計算に入れれば、少し改良できる。

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";   // 質問の書き込み

    char answer = 0;
    cin >> answer;                                   // 回答の読み出し

    switch (answer) {
```

```
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "I'll take that for a no.\n";
            return false;
        }
}
```

switch-statement は、値が一連の定数のどれかと一致するかどうかをテストする。case 定数は別々の値でなければならない。一致する定数がない場合には、**default** が選択される。**default** はなくてもかまわない。

ループを使わずに書けるプログラムはほとんどない。次の例では、ユーザーに数回のチャンスを与えている。

```
bool accept3()
{
    int tries = 1;
    while (tries < 4) {
        cout << "Do you want to proceed (y or n)?\n";  // 質問の書き込み
        char answer = 0;
        cin >> answer;                                  // 回答の読み出し

        switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "Sorry, I don't understand that.\n";
            tries = tries + 1;
        }
    }
    cout << "I'll take that for a no.\n";
    return false;
}
```

while-statement は、条件が false になるまで実行を続ける。

2.3.3 ポインタと配列

配列は、次のように宣言できる。

```
char v[10];      // 10 個の文字の配列
```

同様に、ポインタは次のように宣言できる。

```
char* p;         // 文字へのポインタ
```

これらの宣言の [] は“～の配列”、*は“～へのポインタ”を意味する。配列のインデックス

の下限は 0 なので、v は、v[0] から v[9] までの 10 個の要素を持つ。ポインタ変数は、適切な型のオブジェクトのアドレスを保持できる。

```
p = &v[3];          // pはvの4番目の要素を指す
```

単項の&は、アドレス演算子である。

ある配列から別の配列に 10 個の要素をコピーする方法を考えてみよう。

```
void another_function()
{
    int v1[10];
    int v2[10];
    // …
    for (int i=0; i<10; ++i) v1[i]=v2[i];
}
```

この *for-statement* は、“ i に 0 をセットし、i が 10 未満である間は、i 番目の要素をコピーし、i をインクリメントする ”と読むことができる。インクリメント演算子++は、整数変数を対象とする場合には、単純に 1 を加算する。

2.4 モジュラプログラミング

ここ数年、プログラムの設計の重点は、手続きの設計からデータの構成方法にシフトしてきている。その要因には、プログラムサイズの増加が含まれる。一連の関連し合う手続きとそれに付随するデータをまとめたものは、**モジュール**: *module* と呼ばれることが多い。プログラミングパラダイムは、次のようになった。

必要なモジュールを決めよ。 データがモジュールのなかに隠蔽されるようにプログラムを分割せよ。

このパラダイムは、**データ隠蔽原則**: *data-hiding principle* とも呼ばれる。関連するデータに適用される手続きのグループがなければ、手続き型プログラミングのスタイルで充分である。また、“ 優れた手続き ”を設計するためのテクニックは、モジュール内の個々の手続きに適用できる。モジュールのもっとも一般的な例は、スタック定義だろう。解決しなければならない問題は、次のとおりである。

[1] スタックに対するユーザーインターフェイス（たとえば、push()、pop() 関数）を提供する。
[2] スタックの表現（たとえば要素の配列）には、このユーザーインターフェイスだけを経由してアクセスする。
[3] 初めて使う前にスタックが初期設定されているようにする。

C++は、関連し合うデータ、関数などを別々の名前空間にまとめるメカニズムを提供している。たとえば、Stack モジュールのユーザーインターフェイスは、次のように宣言、利用することができる。

```
namespace Stack {        // インターフェイス
    void push(char);
    char pop();
}

void f()
{
    Stack::push('c');
    if (Stack::pop() != 'c') error("impossible");
}
```

Stack::という限定は、push()、pop() が Stack 名前空間のものだということを示す。これらの名前をほかの用途に使っても、衝突したり混乱の原因になったりはしない。

Stack の定義は、プログラムのなかの分割コンパイルされる部分で提供することができる。

```
namespace Stack {        // 実装
    const int max_size = 200;
    char v[max_size];
    int top = 0;

    void push(char c) { /* オーバーフローをチェックしてcをプッシュする */ }
    char pop()  { /* アンダーフローをチェックしてポップする */ }
}
```

Stack モジュールのポイントは、**Stack::push()**、**Stack::pop()** 実装コードによって、**Stack** のデータ表現からユーザーのコードが切り離されることである。ユーザーは、**Stack** が配列を使って実装されていることを知る必要はないし、実装はユーザーコードに影響を与えずに変更できる。

データは、"隠蔽"したいものの1つに過ぎないので、データ隠蔽の考え方は、簡単に**情報の隠蔽**: *information hiding* の考え方に広げることができる。つまり、関数、型の名前なども、モジュール内に局所化できるということである。そのため、C++では、任意の宣言を名前空間に配置することが認められている（§8.2）。

この **Stack** モジュールは、スタックを表現するための1つの方法に過ぎない。以下の節では、さまざまなプログラミングスタイルの具体例としてスタックを使っていく。

2.4.1 分割コンパイル

C++は、C の分割コンパイルの概念をサポートする。分割コンパイルは、プログラムを部分的に依存しあう一連の断片に組織するために使えるテクニックである。

一般に、モジュールへのインターフェイスを規定する宣言文は、名前から利用目的がわかるようなファイルに配置する。つまり、以下の宣言は、

```
namespace Stack {        // インターフェイス
    void push(char);
    char pop();
}
```

stack.h というファイルに配置され、ユーザーは次のようにこのファイル（**ヘッダファイル**: *header file* と呼ばれる）を**インクルード**: *include* する。

```
#include "stack.h"          // インターフェイスを取り込む

void f()
{
      Stack::push('c');
      if (Stack::pop() != 'c') error("impossible");
}
```

整合性を保つためのコンパイラの仕事を助けるために、**Stack** モジュールの実装を提供するファイルも、インターフェイスをインクルードする。

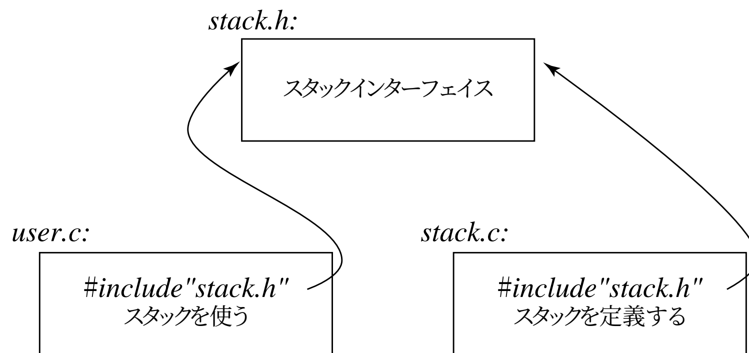
```
#include "stack.h"          // インターフェイスを取り込む

namespace Stack {           // 表現
      const int max_size = 200;
      char v[max_size];
      int top = 0;
}

void Stack::push(char c) { /* オーバーフローをチェックしてcをプッシュする */ }

char Stack::pop()  { /* アンダーフローをチェックしてポップする */ }
```

ユーザーコードは、user.c のような第 3 のファイルに配置される。user.c と stack.c のコードは、stack.h に記述されているスタックインターフェイス情報を共有するが、それ以外の点では 2 つのファイルは独立しているので、分割コンパイルできる。プログラムの個々の部品をグラフィカルに表現すると、次のようになるだろう。



分割コンパイルは、現実のあらゆるプログラムの問題である。**Stack** のような機能をモジュールとして提出するプログラムだけの問題ではない。厳密に言えば、分割コンパイルは、言語の問題ではなく、特定の言語処理系をうまく利用する方法の問題である。しかし、分割コンパイルは、実際の場面で非常に重要な意味を持つ。効果的な分割コンパイルのための最良のアプローチは、プログラムを最大限に細かくモジュール化し、言語機能によってそのモジュール性を論理的に表現し、ファイルを通じて物理的にモジュール化を実現することである。

2.4.2 例外処理

一連のモジュールとしてプログラムを設計するとき、エラー処理もこれらのモジュールの視点から考慮する必要がある。どのモジュールがどのエラーを処理するのか？　エラーを検出したモジュールは、どのような対応をすべきかを知らないことが多い。ある処理を実行しようとしているときに発生したエラーの回復処理は、エラーを発見したモジュールではなく、処理を開始したモジュールによって決まる。プログラムが成長すると、特にライブラリを多用するようになると、エラー（より一般的に言えば、“ 例外的状況 ”）処理の標準が重要になってくる。

再び **Stack** サンプルに戻って考えてみよう。1 字分多く **push()** しようとしたときに**するべきこと**は何だろうか？　**Stack** モジュールの作者には、このようなときにユーザーがしてほしいと思っていることはわからないし、ユーザーは問題を間違いなく検出することはできない（できるなら、最初からオーバーフローは起きないだろう）。答は、**Stack** の実装者がオーバーフローを検出し、（未知の）ユーザーにエラーを報告するというものである。通知を受けたユーザーは、適切な対処をすることができるだろう。たとえば、次のようにする。

```
namespace Stack {          // インターフェイス
      void push(char);
      char pop();

      class Overflow();    // オーバーフロー例外を表現する型
}
```

オーバーフローを検出したら、Stack::push() は、例外処理コードを実行できる。つまり、“ **Overflow** 例外を投げる ”のである。

```
void Stack::push(char c)
{
      if (top == max_size) throw Overflow();
      // cをプッシュする
}
```

throw は、Stack::push() を直接、間接的に呼び出した何らかの関数に含まれている Stack::Overflow 型例外のハンドラコードに制御を渡す。これを実現するために、実装者は、必要に応じて関数呼び出しスタックを巻き戻して呼び出し側のコンテキストを復元する。つまり、**throw** は、マルチレベルの **return** として機能するのである。たとえば、次のコードを見ていただきたい。

```
void f()
{
      // …
      try { // ここで発生した例外は、すぐ下で定義されているハンドラによって処理される

            while (true) Stack::push('c');
      }
```

```
        catch (Stack::Overflow) {
                // まずい、スタックオーバーフローだ。ここで適切に対処する。
        }
        // …
}
```

while ループは、無限にループしようとする。そのため、何度目かの **Stack::push()** 呼び出しが **throw** を引き起こすと、**Stack::Overflow** 例外ハンドラを提供している **catch** 節に制御が移る。

例外処理メカニズムを使えば、エラー処理は従来よりも系統立ったものになり、読みやすくもなる。詳細については、§8.3 と第 14 章を参照していただきたい。

2.5 データ抽象

モジュール化は、大規模プログラムを成功に導くための基本テクニックであり、本書で設計の問題を論じるときには、今後も常に重視され続ける。しかし、前節で記述した形のモジュールでは、複雑なシステムをクリーンに表現するためにはまだ役不足である。本節では、まずモジュールを使って一種のユーザー定義型を提供する方法を示してから、このアプローチが持ついくつかの問題点を解消する方法として、ユーザー定義型を直接定義する方法を示す。

2.5.1 型を定義するモジュール

モジュールを使ったプログラミングを進めていくと、ある型のすべてのデータを型管理モジュールの制御下に置くような一元管理に到達する。たとえば、1 つのスタックを提供する前節の **Stack** モジュールのようなものではなく、たくさんのスタックを使いたいときには、次のようなインターフェイスを持つスタックマネージャを定義することができるだろう。

```
namespace Stack {
        struct Rep;                     // スタックのレイアウトの定義は別の場所
        typedef Rep& stack;

        stack create();                 // 新しいスタックを作成する
        void destroy(stack s);          // s を削除する

        void push(stack s, char c);     // s に c をプッシュする
        char pop(stack s);              // s をポップする
}
```

次の宣言は、**Rep** がある型の名前であると言っているが、型の定義は先延ばしにしている(§5.7)。

```
struct Rep;
```

その次の宣言、

```
typedef Rep& stack;
```

は、“ **Rep** のリファレンス ”に **stack** という名前を与えている。**Stack::stack** をスタックと同一視するとともに、それ以上の詳細はユーザーの目から隠そうという考え方である。

Stack::stack は、組み込みデータ型の変数とよく似た形で動作する。

```
struct Bad_pop { };

void f()
{
      Stack::stack s1 = Stack::create();   // 新しいスタックを作成する
      Stack::stack s2 = Stack::create();   // 新しいスタックをもう１つ作成する

      Stack::push(s1, 'c');
      Stack::push(s2, 'k');

      if (Stack::pop(s1) != 'c') throw Bad_pop();
      if (Stack::pop(s2) != 'k') throw Bad_pop();

      Stack::destroy(s1);
      Stack::destroy(s2);
}
```

この **Stack** は、さまざまな方法で実装できるだろう。ユーザーが実装方法を知らなくてもよいというところが重要である。インターフェイスを変えなければ、**Stack** を実装し直すことにしても、ユーザーは影響を受けない。

たとえば、いくつかのスタック表現データをあらかじめ確保しておいて、**Stack::create()** には未使用のものの参照を渡させるようにする実装が考えられる。この場合、**Stack::destroy()** は、stack 表現データに“ 未使用 ”のマークを付け、**Stack::create()** がそれをリサイクルできるようにすればよい。

```
namespace Stack {                // 表現

      const int max_size = 200;

      struct Rep {
            char v[max_size];
            int top;
      };

      const int max = 16;        // スタックの数の上限

      Rep stacks[max];           // あらかじめ確保してあるスタック表現
      bool used[max];            // stacks[i] が使用中なら used[i] は true

      typedef Rep& stack;
}

void Stack::push(stac s, char c) { /* s のオーバーフローをチェックして c をプッシュ */ }
```

```
char Stack::pop(stack s) { /* sのアンダーフローをチェックしてポップ */ }

Stack::stack Stack::create()
{
    // 未使用のRepをピックアップし、使用中のマークを付け、その参照を返す
}

void Stack::destroy(stack s) { /* sに未使用のマークを付ける */ }
```

ここでしたことは、一連のインターフェイス関数でスタックを表現する型を包み込むことである。このようにして得られた“ スタック型 ”の動作は、これらのインターフェイス関数をどのように定義したか、**Stack** ユーザーにスタックを表現する型をどのような方法で提示したか、スタックを表現する型自体をどのように設計したかの 3 つの要因によって決まる。

このような関係は理想的な状態とは言えない場合が多い。特に大きな問題は、このような“ 偽の型 ”をユーザーに提出する方法が、スタック表現型の詳細によって大きく左右されることである。ユーザーは、スタックをどのような型で表現するかという知識から隔離されていなければならない。たとえば、スタックを表現するためにもっと凝ったデータ構造を使うことにしていたら、**Stack::stack** に対する代入、初期設定のルールは大幅に変わっていただろう。場合によっては、このようなことが実際に望ましいこともある。しかし、便利な **Stack** を提供するという問題は、**Stack** モジュールからスタック表現型の **Stack::stack** に移ってしまった。

さらに、もっと根本的な問題がある。型の実装にアクセスするモジュールを使って作られたユーザー定義型は、組み込みデータ型と同じようには動作しないし、組み込みデータ型ほどのサポートも受けられず、サポートの種類も変わってしまう。たとえば、**Stack::Rep** が利用できるタイミングは、通常の言語規則ではなく、**Stack::create()**、**Stack::destroy()** によって管理される。

2.5.2 ユーザー定義型

C++は、組み込みデータ型と（ほぼ）同じように動作する型をユーザーが直接定義できるようにして、この問題に取り組んでいる。このような型は、**抽象データ型**: *abstract data type* と呼ばれることが多いが、私は、**ユーザー定義型**: *user-defined type* という用語を取ることにしたい。より妥当な**抽象型**の定義のためには、数学的な“ 抽象 ”規定が必要だろう。そのような規定があれば、ここで**型**と呼ばれているものは、そのような本当に抽象的な存在の具体例になるはずである。ここで、プログラミングパラダイムは、次のようになる。

使いたい型を決めよ。 個々の型に対して演算の完全なセットを提供せよ。

ある型のオブジェクトとして必要なものが 1 つだけなら、モジュールを使ったデータ隠蔽スタイルで充分である。

ユーザー定義型の一般例としては、有理数や複素数といった数学型が挙げられる。以下のものを考えていただきたい。

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; } // 2つのスカラー値から複素数を構築する
    complex(double r) { re=r; im=0; }           // 1つのスカラー値から複素数を構築する
    complex() { re = im = 0; }                   // デフォルト複素数:(0,0)

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);      // 2項
    friend complex operator-(complex);               // 単項
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);

    friend bool operator==(complex, complex);        // 等しい
    friend bool operator!=(complex, complex);        // 等しくない
    // …
};
```

complex クラス（ユーザー定義型）の宣言は、複素数の表現形態と複素数に加えられる演算群を規定している。データ表現は *private* である。つまり、**re** と **im** には **complex** クラスの宣言で規定された関数しかアクセスできない。そのような関数は、次のように定義できるだろう。

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}
```

クラスと同じ名前を持つメンバ関数を**コンストラクタあるいは構築子**: *constructor* と呼ぶ。**complex** クラスは、3つのコンストラクタを提供している。1個の **double**、2個の **double** から **complex** を構築するものとデフォルト値の **complex** を構築するものである。

complex クラスは、次のようにして使うことができる。

```
void f(complex z)
{
    complex a = 2.3;
    complex b = 1/a;
    complex c = a+b*complex(1,2.3);
    // …
    if (c != b) c = -(b/a)+2*b;
}
```

コンパイラは、**complex** 値を操作する演算子を適切な関数呼び出しに変換する。たとえば、**c!=b** は **operator!=(c,b)**、**1/a** は **operator/(complex(1),a)** という意味になる。

すべてではないが、ほとんどのモジュールは、ユーザー定義型として表現した方がよい結果が得られる。

2.5.3 具象

ユーザー定義型を設計すれば、さまざまな需要に応えられる。**complex** 型にならって **Stack** 型というユーザー定義型を作ってみよう。話を少し現実に近付けるために、この **Stack** 型は、引数として要素数を取るように定義する。

```
class Stack {
      char* p;
      int top;
      int max_size;
public:
      class Underflow{ };        // 例外として使われる
      class Overflow{ };         // 例外として使われる
      class Bas_size{ };         // 例外として使われる

      Stack(int s);              // コンストラクタ
      ~Stack();                  // デストラクタ

      void push(char c);
      char pop();
};
```

コンストラクタの **Stack(int)** は、このクラスのオブジェクトが作成されるたびに呼び出される。コンストラクタは、初期設定を行う。クラスのオブジェクトがスコープ（有効範囲）から外れたときに終了処理が必要なら、コンストラクタと対になる**デストラクタあるいは解体子、破壊子**: *destructor* を宣言できる。

```
Stack::Stack(int s)        // コンストラクタ
{
      top=0;
      if (10000<s) throw Bad_size();
      max_size = s;
      v = new char[s];      // 自由記憶領域（ヒープ、ダイナミックメモリ）に要素を確保
}

Stack::~Stack()             // デストラクタ
{
      delete[] v;           // 要素の領域を開放して、再利用できるようにする（§6.2.6）
}
```

コンストラクタは、新しい **Stack** 変数を初期設定する。そのために、**new** 演算子で自由記憶領域: free store（**フリー領域**、**ヒープ**: *heap*、**ダイナミックメモリ**: *dynamic memory* とも呼ばれる）からメモリを確保している。デストラクタは、終了処理としてそのメモリを開放する。これらはすべて、**Stack** ユーザーを介在させることなしに行われる。ユーザーは、組み込みデータ型の変数と同じように **Stack** を作成し、利用する。たとえば次のとおり。

```
Stack s_var1(10);                        // 10 個の要素を持つ大域スタック

void f(Stack& s_ref, int i)              // Stack のリファレンス
```

```
{
    Stack s_var2(i);                  // i個の要素を持つ局所スタック
    Stack* s_ptr = new Stack(20);     // 自由記憶領域に確保したStackへのポインタ

    s_var1.push('a');
    s_var2.push('b');
    s_ref.push('c');
    s_ptr->push('d');
    // …
}
```

この **Stack** 型は、命名、スコープ、確保、寿命、コピーなどについて、**int**、**char** などの組み込みデータ型と同じ規則に従う。

当然ながら、メンバ関数の **push()**、**pop()** は、どこかで定義する必要がある。

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    v[top] = c;
    top = top + 1;
}

char Stack::pop()
{
    if (top == 0) throw Underflow();
    top = top - 1;
    return v[top];
}
```

complex、**Stack** などの型は、**具象型**: *concrete type* と呼ばれる。それに対し、インターフェイスが実装のディティールからユーザーをより完全に隔離しているものを**抽象型**: *abstract type* と呼ぶ。

2.5.4 抽象型

モジュールによって実装された“偽の型”(§2.5.1) から本物の型 (§2.5.3) に **Stack** を移行させる過程で、ある特長が失われた。表現がユーザーインターフェイスから切り離されなくなったのである。表現は、**Stack** を使うプログラムコードがインクルードするはずの部分に含まれている。表現は非公開メンバ (*private*) であり、メンバ関数からしかアクセスできないが、そこに存在する。表現が大幅に変更されることがあれば、ユーザーコードの再コンパイルが必要になるだろう。これは、組み込みデータ型と同じように動作する具象型を持つための代償である。特に、型の表現のサイズがわからなければ、ある型の純粋な局所変数を持つことはできない。

型が頻繁に変更されず、局所変数が明確さと効率を持つべき場面では、これは許容できることであり、それどころか好都合でもある。しかし、スタックの実装の変更からユーザーを完全に保護したければ、具象型の **Stack** では不充分である。この問題を解決するには、表現からインターフェイスを完全に切り離し、純粋局所変数を諦めなければならない。

まず、インターフェイスを定義する。

```
class Stack{
public:
    class Underflow{ };        // 例外として使う
    class Overflow{ };         // 例外として使う

    virtual void push(char c) = 0;
    virtual char pop() = 0;
};
```

virtual（**仮想**）という単語は、Simula と C++では、“ ここから派生したクラスで再定義されるかもしれない ”ということを意味する。この **Stack** インターフェイスの実装は、**Stack** から派生したクラスで提供される。**=0** という奇妙な構文は、**Stack** から派生した何らかのクラスが関数を定義しなければならないということを表す。つまり、この **Stack** は、**push()**、**pop()** 関数を実装するクラスに対するインターフェイスとして機能する。

この **Stack** は、次のようにして使う。

```
void f(Stack& s_ref)
{
    s_ref.push('c');
    if(s_ref.pop() != 'c') throw Bad_pop();
}
```

f() が実装のディティールを一切知らないままに **Stack** インターフェイスを使っていることに注意していただきたい。他のさまざまなクラスにインターフェイスを提供するクラスは、**多相型**: *polymorphic type* あるいは**ポリモーフィックな型**と呼ばれることが多い。

実装は、具象型の **Stack** のうち、**Stack** インターフェイスから洩れたものによって構成されるが、そのこと自体は驚くに当たらないだろう。

```
class Array_stack : public Stack {      // Array_stack は Stack を実装する
    char* p;
    int max_size;
    int top;
public:
    Array_stack(int s);
    ~Array_stack();

    void push(char c);
    char pop();
};
```

“ **:public** ”は、“ ～から派生した ”、“ ～を実装する ”、“ ～の下位型である ”と読むことができるだろう。

f() のような関数が実装のディティールをまったく知らずに **Stack** を使えるようにするためには、ほかの関数で **f()** が操作できるオブジェクトを作成しなければならない。たとえば、次のとおり。

```
void g()
{
    Array_stack as(200);
    f(as);
}
```

f() は **Array_stack** の知識を持たず、**Stack** インターフェイスのことしか知らないので、**Stack** のほかの実装もうまく操作できる。たとえば、次のとおり。

```
class List_stack : public Stack       // List_stack は Stack を実装する
    list<char> lc;                    // （標準ライブラリ）文字のリスト（§3.7.3）
public:
    List_stack() { }

    void push(char c) { lc.push_front(c); }
    char pop();
};

char List_stack::pop()
{
    char x = lc.front();              // 先頭要素を取得
    lc.pop_front();                   // 先頭要素を削除
    return x;
}
```

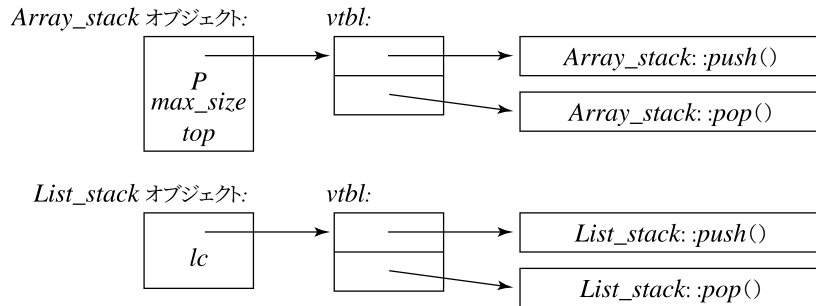
今度の表現は、文字のリストである。**lc.push_front(c)** は、**lc** の第1要素として **c** を追加する。**lp.pop_front()** は先頭要素を取り除き、**lc.front()** は lc の先頭要素を返す。

List_stack を作成し、**f()** にそれを使わせる関数は、次のように書ける。

```
void h()
{
    List_stack ls;
    f(ls);
}
```

2.5.5 仮想関数

f() のなかの **s_ref.pop()** 呼び出しは、どのようにして正しい関数定義に解決される（行き着く）のだろうか？　**f()** が **h()** から呼び出された場合には **List_stack::pop()**、**g()** から呼び出された場合には **Array_stack::pop()** が呼び出されなければならない。このような解決を実現するためには、**Stack** オブジェクトに実行時に呼び出される関数を示すための情報を持たせなければならない。実装テクニックとしてよく使われるのは、コンパイラに **virtual** 関数の名前を関数ポインタテーブルの添字に変換させるというものである。このテーブルは、“仮想関数テーブル”、あるいは単純に **vtbl** と呼ばれる。仮想関数を持つ各クラスは、自分の仮想関数を識別する **vtbl** を持っている。これをグラフィカルに表現すると、次のようになるだろう。



vtbl 内の関数は、オブジェクトのサイズやデータのレイアウトが呼び出し側にはわからないときでも、オブジェクトを正しく利用できるようにする。呼び出し側が知っていなければならないことは、**Stack** のなかの **vtbl** の位置と各仮想関数に対して使われているインデックスだけである。この仮想呼び出しメカニズムは、本質的に“ 通常の関数呼び出し ”メカニズムと同等の効率を持つことができる。スペース上のオーバーヘッドは、仮想関数を持つクラスの 1 つのオブジェクトにつき 1 個のポインタとクラスあたり 1 つの **vtbl** だけである。

2.6 オブジェクト指向プログラミング

データ抽象は、優れた設計の基本要素であり、本書では一貫して設計の重点事項として扱い続ける。しかし、ユーザー定義型だけでは、プログラマの需要に応えられるだけの柔軟性に欠ける。本節では、単純なユーザー定義型の問題点を具体的に示してから、クラス階層を使ってその問題を解決する。

2.6.1 具象型の問題点

具象型は、モジュールを使って定義した“ 偽の型 ”と同様に、一種のブラックボックスを定義する。一度定義されたブラックボックスは、プログラムの残りの部分とは相互に影響を与え合わない。ブラックボックスを新しい用途に使いたければ、定義を変更する以外にない。これはある意味では好都合だが、あまりに柔軟性に欠ける。グラフィックスシステムのために **Shape** という型を定義する場合について考えてみよう。現在のところ、このシステムは円、三角形、正方形をサポートしなければならないものとする。また、次のクラスも持つものとする。

```
class Point{ /* … */ };
class Color{ /* … */ };
```

/*と*/は、それぞれコメントの先頭と末尾を表す。この記法は、複数行に渡るコメントや行末よりも手前で終るコメントで使われる。

さて、ここで次のように **Shape** を定義する。

```
enum Kind { circle, triangle, square };        // 列挙(§4.8)

class Shape {
      Kind k;        // 型フィールド
      Point center;
      Color col;
      // ...

public:
      void draw();
      void rotate(int);
      // ...
};
```

“型フィールド”**k** は、**draw()**、**rotate()** などの演算が対象の図形のタイプを判定するためにどうしても必要である（Pascal 風の言語では、タグ **k** を使って可変型レコードを使うことができる）。**draw()** 関数は、次のように定義できるだろう。

```
void Shape::draw()
{
      switch (k) {
      case circle:
            // 円を描画
            break;
      case triangle:
            // 三角形を描画
            break;
      case square:
            // 正方形を描画
            break;
      }
}
```

これはよくない。**draw()** のような関数は、図形のあらゆるタイプ“についての知識”を持たなければならない。そのため、このような関数のコードは、システムに新しい図形が追加されるたびに増えていく。新しい図形を定義したら、図形に対するすべての演算をチェックし、（恐らくは）変更しなければならない。すべての演算のソースコードにアクセスできるのでなければ、グラフィックスシステムに新しい図形を追加することもできない。新しい図形を追加すると、図形に対するすべての重要演算のコードに“触れる”ことになるので、かなりの技能が必要になるし、他の（古い）図形を処理するコードにバグを導くことになりかねない。汎用型 **Shape** の定義は一般に固定サイズの枠組みを作るはずだが、個々の図形の表現（の少なくとも一部）は、その枠組みに合わせなければならないので、非常に厳しい束縛を受けることになる。

2.6.2 クラス階層

問題は、すべての図形の一般属性（たとえば、色を持つとか描画できるといったこと）と個々の図形タイプの属性（円は半径を持つ図形で、円描画関数によって描画されるといったこと）が区別されていないことにある。オブジェクト指向プログラミングとは、この区

別を表現し、利用することである。オブジェクト指向プログラミングをサポートする言語は、この区別を表現し、利用できる言語要素を持つ言語である。他の言語は、オブジェクト指向プログラミングをサポートしない。

継承のメカニズム（Simula から借りてきた C++の機能）は、この問題に対する 1 つの解法を提供する。まず、すべての図形の一般属性を定義するクラスを規定する。

```
class Shape {
      Point center;
      Color col;
      // …
public:
      Point where() { return center; }
      void move(Point to) { center = to; /* … */ draw(); }

      virtual void draw() = 0;
      virtual void rotate(int angle) = 0;
      // …
};
```

§2.5.4 の **Stack** 抽象型と同様に、呼び出しインターフェイスだけが定義できて、実装がまだ定義できない関数は、**virtual** になっている。特に、**draw()**、**rotate()** 関数は、図形のタイプが決まらなければ定義できないので、**virtual** 宣言されていることに注意していただきたい。

このような定義があれば、図形ポインタのベクタを操作する汎用関数を書くことができる。

```
void rotate_all(vector<Shape*>& v, int angle) // v の要素を angle 度回転する
{
      for (int i=0; i<v.size(); ++i) v[i]->rotate(angle);
}
```

個々の図形を定義するには、それが図形であることを明言して、固有属性（仮想関数を含む）を規定しなければならない。

```
class Circle : public Shape {
      int radius;
public:
      void draw() { /* … */ };
      void rotate(int) {}         // 何もしない関数でよい
};
```

C++では、**Circle** クラスは **Shape** クラスの**派生クラス**: *derived class*、**Shape** クラスは **Circle** クラスの**基底クラス**: *base class* と呼ばれる。**Circle**、**Shape** クラスをそれぞれサブクラス、スーパークラスと呼ぶ用語法もある。派生クラスは、基底クラスからメンバを継承すると言われる。そのため、基底、派生クラスを使うことは、一般的に**継承**: *inheritance* と呼ばれる。

プログラミングパラダイムは、次のとおりである。

> 使いたい型を決めよ。
> 個々の型に対して演算の完全なセットを提供せよ。
> 継承を使って共通点を明示的に示せ。

このような共通点がなければ、データ抽象だけで充分である。2つの型の間で継承と仮想関数を活用できる共通点がどの程度あるかが、ある問題にオブジェクト指向プログラミングを適用できるかどうかのリトマス試験紙となる。グラフィックスの対話的操作のような分野には、明らかにオブジェクト指向プログラミングの莫大な領土が広がっている。それに対し、古典的な算術型やそれに基づく演算には、データ抽象以上のものが必要だとは考えられず、オブジェクト指向プログラミングをサポートするためのメカニズムは不要だと思われる。

システムに含まれる型の間に共通点を見つけ出す仕事は、小さなものではない。引き出せる共通点の量は、システムの設計方法の影響を受ける。このような共通点は、システム設計時、それもシステムの要件を書き出しているときに、積極的に探す必要がある。クラスは、ほかの型を組み立てるための積み木として設計し、既存のクラスのなかに共通基底クラスにまとめられそうな類似点がないかどうかもチェックすべきである。

特定のプログラミング言語の言語要素に頼らずに、オブジェクト指向プログラミングとは何かを説明しようとする試みについては、§23.6の［Kerr, 1987］、［Booch, 1994］を参照していただきたい。

クラス階層と抽象クラス（§2.5.4）は、相互排他的ではなく、互いに相手を補い合う関係を持っている（§12.5）。一般に、ここに挙げたパラダイムは、補い合う性質を持ち、相互に支え合っていることも多い。たとえば、クラスとモジュールは関数を収めているのに対し、モジュールはクラスと関数を収めている。熟練したデザイナは、必要が命じるままにさまざまなパラダイムを活用する。

2.7 ジェネリックプログラミング

スタックを必要とするユーザーが常に文字のスタックを望むとは限らない。スタックは、文字の概念に縛られない一般性を持つコンセプトである。そのため、スタックは文字とは切り離して表現されるべきである。

より一般的に、表現の詳細に縛られずに表現できるアルゴリズムがあり、それが手ごろな労力で論理的に歪んだところなしに実現できるなら、そうすべきである。

プログラミングパラダイムは、次のようにまとめられる。

> 使いたいアルゴリズムを決めよ。
> さまざまな型、データ構造を操作対象にできるように、
> それらをパラメータ化せよ。

2.7.1 コンテナ

文字型のスタックは、スタックをテンプレート化し，**char** という特定の型をテンプレートパラメータに置き換えれば、任意の型のスタックに一般化できる。たとえば次のとおり。

```
template<class T> class Stack {
      T* v;
      int max_size;
      int top;
public:
      class Underflow { };
      class Overflow { };

      Stack(int s);              // コンストラクタ
      ~Stack();                  // デストラクタ

      void push(T);
      T pop();
};
```

template<class T>というプレフィクスは、ターゲットの **T** を宣言のパラメータにする。

メンバ関数も、同様に定義できる。

```
template<class T> void Stack<T>::push(T c)
{
      if (top == max_size) throw Overflow();
      v[top] = c;
      top = top + 1;
}

template<class T> T Stack<T>::pop()
{
      if (top == 0) throw Underflow();
      top = top - 1;
      return v[top];
}
```

これらの定義を用意すれば、次のような形でスタックを使うことができる。

```
Stack<char> sc(200);            // 200 字の文字のスタック
Stack<complex> scplx(30);       // 30 個の複素数のスタック
Stack< list<int> > sli(45);     // 45 個の整数リストのスタック

void f()
{
      sc.push('c');
      if (sc.pop() != 'c') throw Bad_pop();

      scplx.push(complex(1,2));
      if (scplx.pop() != complex(1,2)) throw Bad_pop();
}
```

同様に、リスト、ベクタ、マップ（連想配列）などもテンプレートとして定義できる。任

意の型の要素のコレクションを格納するクラスを**コンテナクラス**: *container class*、あるいは略して**コンテナ**: *container* と呼ぶ。

テンプレートは、テンプレートを使ったからといって“ 手書きのコード ”と比べて余分な実行時オーバーヘッドを生まないようにするためのコンパイル時メカニズムである。

2.7.2 汎用アルゴリズム

C++標準ライブラリはさまざまなコンテナを提供しており、ユーザーも自分自身の独自コンテナを書くことができる（第3、第17、第18章）。そこで、アルゴリズムの対象となるコンテナをパラメータ化することによって、ジェネリックプログラミングパラダイムをもう1度適用することができる。たとえば、コンテナごとに **sort()**、**copy()**、**search()** を書かずに、**vector**、**list**、配列をソート、コピー、探索できるようにしたい。また、単一のソート関数が受け付けられるように特定のデータ構造にデータを変換することも避けたい。コンテナがどのようなタイプかを知らずにコンテナを操作できるようにするためには、コンテナ定義の一般的な方法を見つけ出す必要がある。

そのための方法として、C++標準ライブラリでコンテナや非数値アルゴリズムが採用しているのは、シーケンス:sequence と呼ばれる概念に焦点を当て、反復子によってシーケンスを操作する方法である。

シーケンスの概念をグラフィカルに表現すると、次のようになるだろう。

シーケンスは、先頭と末尾を持っている。反復子は1つの要素を参照し、反復子にシーケンスのなかの次の要素を参照させるための演算を提供している。シーケンスの末尾は、シーケンスの最後の要素の次を参照する反復子である。“ 末尾 ”の物理的な表現は、番兵要素でもかまわないが、そうである必要もない。実際、ポイントは、このシーケンスという概念がリスト、配列を含め、さまざまな表現をカバーすることにある。

“ 反復子を介した要素へのアクセス ”とか“ 反復子に次の要素を参照させる ”といった演算には、標準的な記法が必要である。間接参照演算子の*に“ 反復子を介した要素へのアクセス ”の意味を持たせ、インクリメント演算子の++に“ 反復子に次の要素を参照させる ”の意味を持たせるのは、（アイディアが理解できたら）当然の選択だろう。

すると、次のようなコードを書けるということになる。

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
    while (from != too_far) {
        *to = *from;          // 反復子が指す要素をコピー
        ++to;                 // 出力の次の要素
        ++from;               // 入力の次の要素
    }
}
```

このコードは、正しい構文とセマンティクスを持つ反復子を定義できるあらゆるコンテナをコピーする。

C++組み込みの低水準配列、ポインタ型は、反復子の要件を満たす演算を持っているので、次のように書くことができる。

```
char vc1[200];          // 200字の配列
char vc2[500];          // 500字の配列

void f()
{
     copy(&vc1[0],&vc1[200], &vc2[0]);
}
```

このコードは、**vc1** の第 1 要素から最終要素までの全要素を **vc2** の第 1 要素の位置から順に **vc2** にコピーする。

標準ライブラリのすべてのコンテナ（§16.3、第 17 章）は、この反復子とシーケンスの概念をサポートする。

テンプレートのパラメータ、**In**、**Out** は、単一の引数ではなく、ソースとターゲットの型を示すために使われている。このような構文が選ばれたのは、あるタイプのコンテナから別のコンテナへのコピーが必要なことがよくあるからである。たとえば、次のとおり。

```
complex ac[200];

void g(vector<complex>& vc, list<complex>& lc)
{
     copy(&ac[0],&ac[200],lc.begin());
     copy(lc.begin(),lc.end(),vc.begin());
}
```

このコードは、配列を **list** に、**list** を **vector** にコピーしている。標準コンテナでは、**begin()** は、第 1 要素を指す反復子である。

2.8 最後に

完璧なプログラミング言語はない。幸い、プログラミング言語は、完璧でなくても、大システムを構築するための優れたツールになれる。実際、汎用プログラミング言語は、それが使われるさまざまな仕事のすべてに対して完璧であることはできない。ある分野で完璧なものは、ほかの分野では致命的な欠陥を持つことが多い。ある分野で完璧であるということには、専門化という意味が含まれるのである。そこで、C++はさまざまなシステムを構築するための優れたツールになるように、またさまざまなアイディアを直接的に表現できるように設計された。

言語の組み込み機能ですべてのことが直接表現できるわけではない。実際、それは都合のよいことでもない。言語の機能は、さまざまなプログラミングスタイルとテクニックをサポートするために存在する。そのため、言語の学習では、言語のすべての機能のすべての詳細を理解することではなく、その言語本来の自然なスタイルをマスターすることに重

点を置く必要がある。

現実のプログラミングでは、言語のもっとも知られていない機能を知っていることや最大数の機能を使っていることは、ほとんど役に立たない。1つの言語機能をほかから切り離して知っていても、ほとんど意味はない。テクニックとほかの機能がもたらす文脈のなかでしか機能は意味を持たないし力を発揮しない。そこで、以下の章を読むときには、C++のディティールを検討している真の目的は、適切な設計の文脈のなかで優れたプログラミングスタイルをサポートするためにそれらの機能を連携させて使えるようにするためだということを忘れないようにしていただきたい。

2.9 アドバイス

[1] パニックに陥らないように！　すべてはいずれ明らかになる: §2.1
[2] 優れたプログラムを書くために C++のすべてのディティールを知る必要はない: §1.7
[3] 言語の機能ではなく、プログラミングテクニックを重視せよ: §2.1