論理的にC++でプログラムを書こう

青木　健一郎

まえがき

現代人はPC，スマートフォン，タブレットなどでアプリケーションを常時使っています．これらのアプリケーションの中身は誰かが造ったプログラムです．通常は意識しないかも知れませんが，これらのアプリケーションを走らせるOS（オペレーティングシステム）もプログラミングで創られています．そして，日常的に接する自動車，洗濯機，炊飯器等のあらゆる器材にもプログラムが組み込まれています．これらはどのように造られ，どのような仕組みで動いているのでしょうか？

アプリケーション等のプログラミングされた物の仕組みを理解するには，自分でプログラミングを経験することが必要です．たとえば，泳ぐ体験をしないと，水泳はわからないでしょう．本書では，C++での実践的なプログラミングを初歩から解説しています．実用的なプログラミングをするために基盤となる基本的概念は含めたつもりです．本書を読めば，様々なプログラミング・プラットフォームの仕様（APIとよびます）を理解できるようになるはずです．また，C++言語を越え，プログラミングとは何であるかを実感し，現代の本格的なプログラミングに必要な基礎を理解を得られることも目的としています．

C++はC言語を進化させた言語で，最もよく開発現場で使われている言語の1つです．高い実用性を持つことが特徴です．CとC++は様々なOS，表計算，ワープロ，ゲーム等の商用アプリケーション，ソフトウェアから，Arduino等の組み込みプログラミングまで用いられています．このように，抽象的なレベルからハードウェアに近いレベルまでのプログラミングをカバーできる言語として秀でています．C++はCよりもプログラムを書きやすい言語であるからこそ，多くの場面で使われています． C ++を学ぶ前にCを勉強する必要はありません．

本書はプログラミング経験があることは仮定していません．簡単な例を通じて基本概念を説明し，それを発展させて理解を深め，広げる形式をとっています．必要な概念を適宜導入します．プログラミングに必要なのは，論理的な考え方だけです．数学や自然科学が得意である必要はありません．自分でプログラムを書き，それをいろいろ改変してみて，理解を深めて下さい．本書は読者がC++でプログラミングすることを通じてC++言語の使い方を理解することを想定しています．オブジェクト指向，ポインタ，等の様々な概念は自分でプログラミングをし，その説明と組み合わせることでわかるでしょう．

プログラミングは，自分で実践して初めて理解できるものだと思います．できるだけ早くに実用的，あるいは実用につながるものを書ける，そして，様々なツールを使えることが重要だと考えています．使いながら，次第に文法についての理解も深めていけます．その観点から，基本的な概念をコンパクトにまとめ，文法の理解を確かめる簡単な例を含めています．必要に応じて，様々な概念，文法を次第に導入しています．簡単な例は実用性があまりなく，面白くないかも知れませんが，章の最後の節に，応用例を含めています．文法の構造の仕組み，そしてなぜそのようなプログラムの書き方をするのかも論理的に説明するようにしています．本書は必要な要素は含めてコンパクトにし，できるだけ早く実践にたどり着けることを目的としています．実践的にプログラミングをする際に，助けとなるであろうと思う点も随所で説明し，コラムとしても含めました．

プログラミングは「ものづくり」で創造的な活動です．ぜひ皆さんもプログラミングを楽しんで下さい．

2020年
青木健一郎

Contents

1 プログラミング，C++言語とは **1**
1.1 プログラミングとアプリケーション 1
1.2 C++言語を使ってみよう 3
1.3 日本語の扱いについて 5

2 変数 **7**
2.1 変数と型 7
2.2 代入の意味 8
2.3 変数の型 8
2.4 コンソール入力 9
2.5 コメント 10
2.6 演算 11

3 関数 **15**
3.1 関数の構造 15
3.2 様々な関数 16
3.3 デフォルト引数 17
3.4 関数のオーバーロード 18
3.5 `main`関数 20
3.6 関数内の変数は外では使えない 21
3.7 関数：応用 22

4 クラスとオブジェクト **25**
4.1 クラス，オブジェクトとは 25
4.2 宣言と定義 27
4.3 コンストラクタ 28
4.4 コンストラクタのオーバーロード 29
4.5 コンストラクタにおけるデータ初期化 31
4.6 `const`修飾子 31
4.7 デストラクタ 32
4.8 なぜクラスを使うのか — プログラミング法の進化 33
4.9 `struct` 34
4.10 クラスとオブジェクト：応用 34

5 流れの制御 **36**
5.1 `if` 36
5.2 条件 36
5.3 `if`, `else if`, と `else` 39
5.4 `while` 42
5.5 `break` 43
5.6 `for` 44
5.7 `do while` 45

5.8 switch . . . 46
5.9 流れの制御：応用 . . . 49

6 配列，ポインタとvector 51
6.1 配列 . . . 51
6.2 配列と動的メモリ確保 . . . 52
6.3 ポインタ . . . 53
6.4 オブジェクトへのポインタ . . . 55
6.5 new/delete . . . 55
6.6 配列の正体 . . . 58
6.7 配列の new，delete . . . 59
6.8 vector . . . 59
6.9 vectorと動的メモリ . . . 61
6.10 vectorの初期化 — コンストラクタ . . . 62
6.11 関数の値渡しと参照渡し . . . 63
6.12 vector：応用 . . . 65

7 ファイルと入出力 68
7.1 ファイルへの出力の基本 . . . 68
7.2 ファイルからの入力 . . . 69
7.3 ファイル入出力のポイント . . . 70
7.4 ファイル入出力：応用 . . . 72

8 継承 76
8.1 継承の目的と概念 . . . 76
8.2 継承の基本 . . . 77
8.3 継承とvirtual関数 . . . 78
8.4 継承とコンストラクタ . . . 79
8.5 継承とポインタ . . . 81
8.6 ポリモーフィズム（多態性） . . . 82
8.7 抽象クラス . . . 85
8.8 継承：応用 . . . 86
8.9 継承を使うのか使わないのか . . . 88

9 様々な文法のポイント 91
9.1 テンプレート . . . 91
9.2 STL . . . 93
9.3 演算子のオーバーロード . . . 96
9.4 mainとargc，argv . . . 98
9.5 ブロックとスコープ . . . 98
9.6 自動変換 . . . 100
9.7 コンパイラ，プリプロセッサ，ヘッダファイル . . . 102
9.8 関数オブジェクト（ファンクタ） . . . 105
9.9 ラムダ式（無名関数） . . . 108
9.10 auto . . . 111

10 最後に 114

11 プログラミングやC++についてさらに学びたい方へ 115

A コンパイル，プログラム実行，とIDE 117

Chapter 1

プログラミング，C++言語とは

1.1 プログラミングとアプリケーション

表計算，ワードプロセッサ，ゲームなどはコンピュータ上の**ソフトウェア**の例です．ソフトウェアは，このような場合には，このようにする，といった，与えられた**指示**を実行します．このような指示をまとめたものは**プログラム** で，それを創る作業を**プログラミング**とよびます．コンピュータの機械そのものは，**ハードウェア**で，その上で走るソフトウェアが無ければ，漬物石と同じです．表計算，プレゼンテーション作成用ソフトウェア，ワードプロセッサ，ゲーム，ウェブブラウザなどの特定の目的を持ったソフトウェアを**アプリケーション**， あるいはアプリケーションソフトウェア，アプリ，appとよびます．

コンピュータ上でアプリケーションをダブルクリック等で呼び出しますが，その呼び出しをして，アプリケーションを走らせて管理するのが**OS**（オペレーティングシステム）です．Windows，macOS，iOS（iPhone等のOS)，Linux，Android（多くのスマートフォンのOSでLinuxの一種）などが典型的なOSの例です．

コンピュータは指示通りにしか作動しません．そして，コンピュータは**機械語**（マイクロコードともよびます）しか理解しません．この機械語はハードウェアの機種によって異なります．

通常は，機械語ではなく，より人間に理解しやすい「高レベル」[1]なC++，Java，Ruby，Pythonなどの

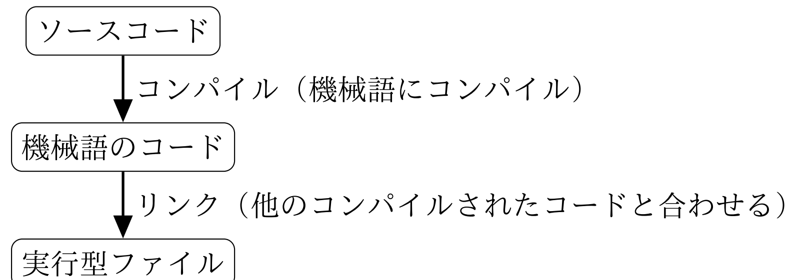


Figure 1.1: プログラムを作動させるまで.

言語（プログラミング言語）でコード（あるいはソースコード）を書きます．コードは人間にわかるように書かれたコンピューターへの指示書です．そのコードを機械語に翻訳して，機械に指示を出すことになります．コードを機械語に翻訳したものが**実行形式**のプログラムです．より正確には，図1.1に示したように，一般的にはプログラムを翻訳し，別に用意されているソフトウェアを必要に応じて含め，実行形式のプログラムを作ります．プログラムを翻訳することを**コンパイル**，翻訳するソフトウェアを**コンパイラ**とよびます．一般には複数のコンパイルされたコードと繋ぎ合わせて最終的なプログラムを作ります．この繋ぎあわせ作業を**リンク**とよびます．ここで説明した全体の作業過程を意識する必要がある場合は通常少ないかも知れません，しかし，トラブルが起きた場合に原因を突き止めるためには流れの理解は必要です．

実行形式のファイルはハードウェア，OS等に依存しますが，C++，Java，Fortran，Ruby，Perl，Python等のプログラミング言語で書かれたプログラムは依存しません．プログラミング言語にはいろいろ

[1] ソフトウェアでは，より抽象度が高く，論理に近いことを「高レベル」，機械に近いことを「低レベル」とよびます．

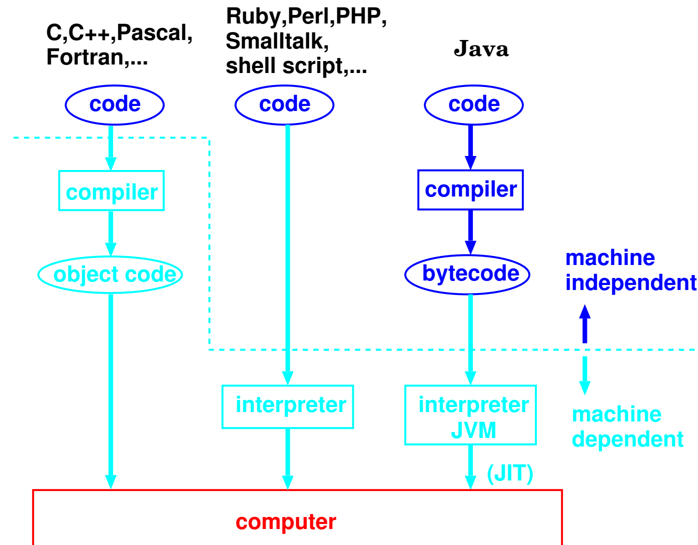


Figure 1.2: コンピュータ言語とプログラムの実行.

な特徴がありますが，プログラミング言語によって，プログラムの実行のしかたが少し異なります．大まかに分けると，次の2つに分けられます（(図1.2参照)）.

A 指示を与えると，それをその場で翻訳しつつ実行するプログラミング言語（Ruby，Perl，Python，シェルスクリプト，等）．スクリプト系言語ともよびます.

B プログラムを，機械語に翻訳した実行形式のファイルを作り，それを実行するプログラミング言語．C,C++,Fortran言語などがその例です.

実行形式のプログラムを使うには翻訳（コンパイル）する手間はかかりますが，プログラムの作動が速い利点があります．よって，画像処理アプリケーション，オフィス系アプリケーション，ウェブブラウザ，ゲーム，などPC上で使う多くのアプリケーションは実行形式のプログラムです．ただ，実行形式のファイルは機種に依存するので，様々なハードウェア，OS，で動かすためにはそれぞれのためにコンパイルしておく必要があります.

上のA，Bの中間的言語もあり，Javaがその典型例です．仮想的なコンピュータ（Java Virtual Machine，JVM）の機械語に翻訳した実行形式のファイルを作り， JVMがプログラムを実行します．JVMは仮想的でコンピュータのハードウェアに依存しないので，機種，OSごとにプログラムはコンパイルする必要が無い利点があります．一方，プログラムを走らせるためには，JVMをOS，機種ごとに実装しておく必要があります.

プログラミングのポイント プログラミングは実技なので，実践無しで理解はできません．たとえば，水泳の本を読んだりビデオを見たりしても，それだけでは泳げないのと同様です．プログラミングでは人間である自分の時間が一番大事だと考えて下さい．そのためには，プログラムを書くのが楽であることが重要です.

プログラムを書くには，まず，プログラムで達成したい内容を論理的に整理する必要があります．それにより，間違いが減り，再利用しやすくなります．その自分の整理した論理の流れに沿ってコーディングをすべきです．そして，プログラムが動かなかったり，意図通りに作動しない時には，闇雲にいろいろ試したくなる時もあるでしょうが，冷静に論理的に分析する必要があります．多くの場合，「急がば廻れ」で，落ち着いて考える方が効率的です．コーディングを勉強する際は，確認を含めいろいろ試してみるべきです．どんなコードを書いて実行してもコンピュータは壊れません.

プログラミングでの重要なポイントは以下の2点です.

1. **プログラムが意図通りに作動する**
2. プログラムがシンプル

特に1点目が圧倒的に重要です．どのように華麗なコードを書こうが，まず動いて，しかも意図通りに作動しなければ意味はありません．そして，コードは可能な限りシンプルであるべきです．その方が，間違いをしにくく，再利用しやすく，そしてコンピュータが実行効率も良い場合が多いです．「シンプル」とは何を意味するかは，おいおい説明していきます．そして，経験を通じてわかっていくでしょう．また，意図通りに作動するコードを書くためには，間違いを極力しにくいように書くようにします．これについても後で説明します．

1.1.1 プログラム実行までの流れ

ここでプログラミングに関する概念と言葉を整理しておきましょう．

- プログラミング言語で書かれた指示書をソースコード，あるいは単にコードとよびます．以下ではコードとよびます．
- コードを書く作業がプログラミングで，コーディング，ともよびます．この作業には，書いたものを正しく実行できるように，直す作業も含まれます．
- コードをコンパイルしたものを実行することを，以下ではプログラムを実行するといいます．

1.1.2 プログラミングは難しくない！

プログラミングでは簡単な部品を**確実に**組み合わせて作っていきます．複雑な機能もこれで実装できます．複雑な，あるいは大掛かりなプログラムを書くのに必要なのは，まず，実行しようとしている内容を論理的に整理することです．あとは，その論理に合わせて各部品を確実に創り，組み合わせることです．各部品は難しくありません．複雑さは，論理の複雑さや，機能の多さからくる作業の量にあります．ただし，1つ1つの部品は正確に作る必要があります．コンピューターは気がきかないので，ちょっとしたミス，スペルミスも，文法ミスも見逃してくれません．わかってくれるだろう，と思ってもわかってくれません．よって，プログラミングでは，基本的で簡単な事を正確にできることが非常に重要です．これを気に留めておけば，楽しくプログラミングできるはずです．

1.1.3 C++言語の特徴

C++はプログラミング言語の１つで，あらゆるOS上で使えます．C++の特徴の1つは，抽象的なプログラミング（「高レベルのプログラミング」）からハードウェアに近いプログラミング（「低レベルのプログラミング」）までできる事です．また，非常に実用的で，現場で最も多く使われている言語の１つです．現に，オフィス系ソフトウェア，ウェブブラウザ，ゲーム，数値計算，等あらゆる種類のソフトウェアで多く使われています．調べてみれば実感できるはずです．特にパフォーマンスを要求される状況では威力を発揮します．OS,組み込み等のハードウェアに近い状況でも多くの場面で使われています．

このように汎用性が高く，実用性も高く，多用されているのは**使いやすいプログラミング言語**だからです．元は，カーニハンとリッチーがOSを書くためにC言語を1972年に開発しました（実際，Windows，MacOS，Linux等，現在のOSはほとんどの部分がC言語で書かれています）．C言語は多くの場面で使われた，そして現在も使われているプログラミング言語です．それにオブジェクト指向（OOP，Object Oriented Programming)を取り入れ，進化させたのがC++言語です．ストロヴストルップが開発し，当初は「オブジェクトを含んだC」（“C with objects”）とよばれていました．C++言語はその経緯から，C言語を実質的に含んでいます．C++言語がとっつきにくいかも知れない1つの理由は，C言語を含み，多くの機能を重複して含んでいて，言語の仕様が大きい事です．C++言語を学ぶ際には，全てを理解しようとせずに，必要なものをしっかり把握して使っていく姿勢が良いと私は考えています．この方針に沿って以下説明していきます．

1.2 C++言語を使ってみよう

1.2.1 基本の基本

初めてのC++プログラム 以下では簡単な具体例を通じて，C++言語でのプログラミングを学んでいきます．

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

int␣main(){
␣␣cout<<"Hello␣World!"<<endl;
}
```

← コンソールに文字列出力を指示

今回だけは空白を␣で明示しました（以下では表示しません）．これをコンパイルして走らせてみましょう（コンパイルの仕方，プログラムの走らせ方は付録Aにまとめてあります）．コンソールに

```
Hello World!
```

と表示されるはずです．コンソール（ターミナルともよびます）はテキスト入力，出力を通じてコンピュータとやりとりをする画面です．[2]WindowsのPowerShell，コマンドプロンプトや MacOSのTerminalはコンソールの例です．まず，C++言語の文法の詳しい解説の前に，この例を発展させて，プログラミングの感じを掴んでみましょう．

コードの構造は以下のとおりです．

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

int main(){
  指示1;
  指示2;
  ...
}
```

ここでは指示の部分だけを説明し，それ以外は後に説明します．上の例では指示は1つ（`cout<<"Hello World!"<<endl;`）しかありませんでした．以下では，指示の部分を書き換えてみて何が起きるかみてみましょう．後により詳しく説明しますが，最低限の文法をあげます：

- ソースコードのファイルはテキストファイル（ファイルのデータが文字だけ）です．ワープロや表計算のファイルではありません．どのエディタで編集しても良いですが，C++の文法に対応して色分け等をしてくれるエディタを使った方が楽です．
- ファイル名は任意です．たとえば，`source1.cpp`のように，拡張子は通常`.cpp`を用います（拡張子`.cc`を使う場合もあります）．
- 引用符（"）にはさまれている部分は**文字列**（文字が並んだデータ）です．
- コードは，文字や文字列とコメント以外は**全て半角の文字，数字，記号**で構成されます（コメントについては後に説明します）．特に誤って全角スペース入れると気づきにくいので注意して下さい．
- 小文字と大文字は区別します．
- 改行や空白は多く含めても同じ指示です．たとえば，`using namespace` ⋯より`<<endl;` まで全て1行に書いてもコンピュータにとっては同じです（`#`で始まる行は特殊なので後に説明します）．
- `cout`は標準出力（コンソール出力）で，`<<`は次に続くものを出力する指示です．`endl`は行の終わりです．つまり`cout<<"Hello World!"<<endl;`はコンソールに文字列`Hello World!`を出力し，行を終わらせる（改行する）指示になります．

  試しに，`<<endl`を無くして，コンパイルして走らせてみて下さい．このように，いろいろ試して，何が起きるかを調べることは勉強になります．

[2]コンソールのように，文字を通じてコンピュータとやりとりするインターフェースをCUI（Character User Interface)とよびます．これに対し，ボタンをクリックしたり，マウスでなぞったりする文字以外の要素を含むインターフェースをGUI（Graphical User Interface）とよびます．

- 上の例でもそうですが，1つの指示内に複数の出力（<<）を含めても良いです．逆に，分けて，`cout<<"Hello World!"; cout <<endl;`と書いても上と同じ指示になります．

いろいろプログラムを書いてみよう では，指示を書き換えてみましょう．

- <<を用いて，文字，文字列や数字（整数，実数）を出力できます．
- 四則演算は+-*/で表します．そして，整数の割り算の余りは%で表します．

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

int main(){
  cout<<"計算します："<<endl;
  cout<<"2+3="<<2+3<<endl;
  cout<<"2.5*3="<<2.5*3<<endl;
}
```

上のコードをコンパイルして実行すると以下の出力が得られます．

```
計算します：
2+3=5
2.5*3=7.5
```

まず，文字列「計算します :」を出力して改行し，そして文字列2+3=を出力し，2+3（当然これは5です）を出力し，改行しています．掛け算についても同様です．整数だけではなく，小数点を含んだ数も使えることに注意しましょう．文字列のデータ"2+3="と計算の指示の2+3の違いを意識して下さい．様々な計算式を，四則演算を使ってコーディングして走らせて確認してみましょう．

1.3 日本語の扱いについて

1.3.1 コードは1バイト文字で

日本語の扱い方はC++の文法とは厳密には別問題ですが，コード内で日本語を扱う場合は出てくるので，説明します．

C++のコードは文字列の中身以外は英数字（a,b,c,$\cdots$,z， A,B,C,$\cdots$,Z， 0,1,2,$\cdots$,9），句読点（, . : ;），空白（␣），記号（<,>,-,#,",',_）等，**全て1バイト文字**で書きます．1バイト文字は半角文字ともよびます．1バイトはデータ量の単位で，8ビットを意味します．1ビットは2種類のデータ（通常0，1とします）を保持でき，1バイトは$2^8=256$種類のデータを表現できます．アルファベットは26文字で，大文字，小文字，そしてその他記号等を含めても1バイト文字で表現できます．コメント部分は1バイト文字に制限されません．（コメントについては2.5節で説明します）．

1.3.2 日本語とマルチバイト文字

日本語の文字は，漢字を含めると種類が多すぎて1バイト文字では表現できません．よって，2バイト文字（全角文字とよぶこともあります）で表現します．2バイト以上の文字をマルチバイト文字とよびます．よって，上で説明したように，コード内では日本語文字は文字列とコメント以外では使えません．少し紛らわしいのは，2バイト文字にも英数字や空白があることです．たとえば，1バイト文字のb,0に対し，2バイト文字のｂ，０があります．２バイト文字はコードで使えないので注意しましょう．特に，2バイトの空白（全角スペース）は通常見えないので，コードに紛れ込ませてしまうと，わかりにくいです．全て文法的に正しいはずなのに，変なエラーが出てコンパイルできない場合は，2バイト文字，特に空白がコードに紛れ込んでいないことを確認しましょう．

1.3.3 文字化け

日本語表示のはずなのに読めない「文字化け」の現象を経験したことのある人は多いでしょう．1バイト文字では文字化けは起きません．1バイトを0 ~ 255で表現すると，各数字がどの文字に対応するかについての世界共通の標準があるからです（たとえば，101はA，72は:，等々）．日本語文字にも番号と文字の対応の標準があり，この対応を**エンコーディング**とよびます．ただ，この標準は複数あります．間違った標準を使って，数字から文字に直すと「文字化け」が起きます．エンコーディングは，2019年現在，macOS, Linux等のWindows以外のOSではUTF-8が主流です．WindowsではS-JIS（Shift JIS，あるいはCP932とよぶこともあります）を用います．「文字化け」が起きる場合は，どのエンコーディングを使っているか確認して下さい．わからなくなってしまった場合は，文字列をふくめ，1バイト文字のみを用いてコーディングすると良いでしょう．

1.3.4 S-JISと「ダメ文字」問題

S-JISの文字列を扱う場合には，「ダメ文字」に注意する必要があります．細い説明は省きますが，構，ソ，能，噂などの比較的よく使ういくつかの文字は文字列でそのままでは表現できません．コンパイルエラーになったり，表示がおかしくなったりするので，S-JISを使う際には頭の隅に留めておかないと混乱することがあります．ダメ文字を使いたい場合は，たとえば`cout<<"構造";`とは書かずに`cout<<"構\造";`のように\をダメ文字の後に加えることで回避できます．UTF-8にはダメ文字の問題はありません．

まとめ：プログラミングとC++

- プログラム：コンピュータへの指示書.
- アプリケーション：特定の目的を達成するソフトウェア.
- （ソース）コード：プログラミング言語で書かれた指示書．文字列，コメント以外は全て1バイト（半角）文字.
- コンパイル：コードを機械語に翻訳する作業.
- C++言語：OS,ハードウェアに依存しないプログラミング言語の1種.
- C++言語の特徴：実用性，汎用性.
- 2バイト文字，マルチバイト文字，全角文字：日本語の文字に使う.
- 1バイト文字，半角文字：英数字（記号含む）.

Chapter 2

変数

本章では，変数を用いてプログラムを書くことを学びます．

2.1　変数と型

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

int main(){
  int n;              ← int型変数nを宣言
  n = 100;            ← nに100代入
  cout<<n<<endl;      ← coutにnの値を出力
}
```

このプログラムの実行結果は，以下のとおりです．

```
100
```

変数はデータを保持でき，以下ではそのデータの値[1]を**変数の値**とよびます．これから説明するように，変数は保持できる値の種類によって，様々な**型**があります．上の例では**int型**（整数，integerの型）**変数n**を**宣言**し，nに100を**代入**しています．一般に変数は次のように宣言します．

```
型名 変数名;
```

int型の変数は整数を保持し，<<では値を出力します（この場合は100）．上の例でcout<<n<<endl;のかわりに，cout<<"n"<<endl;と書くと，

```
n
```

が出力されます．これは文字列としてのnを出力するからです．注意しましょう．

変数は使う前に必ず型を宣言する．

これは絶対に忘れないで下さい．さらに，1つの変数について宣言するのは1回だけです．2回以上すると文法エラーになります．原則として，変数を**初めて使う直前に宣言する**のがコードがわかりやすく，間違えにくいのでお勧めです．全変数をコードの初めで宣言する必要は無く，それは勧めません．

変数について，他に留意すべき点をあげます．

1. 変数名はアルファベット大文字，小文字，数字，_の組み合わせです．1文字目には数字は使えません．たとえば，suuji，AaBa，n1，N1等は変数名に使えますが，1nは変数名として使えません．

[1] 値は数字とは限りません．たとえば文字列も変数の値の一種です．

2. 大文字，小文字は区別されます．たとえば，変数 X1 と x1は異なる変数として認識されます．

3. =は代入を意味し，数学の=とは意味が異なります．よって，n = 100;と100 = n;は同値に見えるかも知れませんが，後者は文法違反です．定数である100に値を代入できないからです．

4. int n; n = 100; の2行は int n=100;，あるいは int n(100); と1つの指示にまとめても良いです．2番目の書き方はわかりにくいかも知れませんが，4.3節で説明します．

5. 上のコードで=，<<の前後に空白を入れる必要は文法的には無いですが，読みやすいように空白を入れると良いでしょう．

main，include，namespace　上のコードで指示をint main(){ }でくくっているのは今の段階では「おまじない」と思っていて下さい（3.5節参照）．#include <iostream>では入出力関係の既に定義されているものを読み込んでいます．使うデータ，関数等によって読み込むものは異なります．using namespace std;によって変数や指示を省略形で楽に書けるようにしています．これらについては3.5節でより詳しく説明します．

2.2 代入の意味

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

int main(){
  int n = 100;          ← int型変数nを宣言し値は100
  cout<<n<<endl;        ← coutにnの中身を出力
  n = n+10;             ← nにn+10代入
  cout<<n<<endl;        ← coutにnの中身を出力
}
```

このプログラムを実行すると，以下の結果が得られます．

```
100
110
```

前節ポイント4.で説明したように，n の宣言と値100の代入は1行で行いました．ポイント3. で説明したように，=は代入なので，n = n+10; はnにn+10の値を代入するという意味です．数式として考えると矛盾があるので，数式との差がはっきりわかるでしょう．

2.3 変数の型

2.3.1 よく用いる変数の型

まずここではよく用いる変数の型を説明します．コンピュータは有限な能力しか持たないので，限界があることも意識しておく必要があります．

int型　変数の値は整数です．-2147483648から2147483647までの整数値をとれます．

double型　実数型変数です．-1.7×10^{308}から-1.7×10^{308}の範囲の実数を15桁の精度で保持できます．

string型　文字列型変数です．

void型　文字通り「無し」の変数です．使い方がわからないかも知れませんが，以下で実例が出てくるので，そこで理解しましょう．

　コンパイラによっては，上で示したよりもデータの保持できる領域が広かったり，精度がもっと高い場合もあります．

2.3.2 その他の型

- char：文字（1バイト文字）1個の変数．
- wchar_t：2バイト以上のマルチバイト文字の変数．
- bool：ブール代数型変数．真（true）か偽（false）の値を取る変数．
- float：実数型変数で単精度．doubleは倍精度の実数なので，floatの方が精度が低く，値の取る領域が狭いです．

修飾語　さらに，データ型に以下の修飾語を付けられる場合があります．

- short：データ量を小さく．
- long：データ量を大きく．
- signed：符号付き．
- unsigned：符号無し．

たとえば，long int，short intのような型を使えます（使えない組み合わせもあります）．この場合，整数型データの大きさ（よって範囲の大きさ）は大きい方からlong int，int，short intとなります．コンパイラの実装に依存しますが，大きさが等しい場合も含まれている事に注意してください．unsigned intは正の値（0を含む）しか取れない変数の型です．このような変数では値が負の場合は，想定した値を取らないので注意が必要です．

データの大きさを指定した整数型　64ビットの整数の型，int64_t，などのように，データの大きさを指定した整数の型もあります．64ビットの符号なしの整数の型はuint64_tです．8，16，32，64ビットの整数（符号あり，無し）の型があります．最低ビット数を指定したint_least8_t，uint_least8_t等（これらの例では8ビット以上）もあります．通常は使う必要は無いですが，既存のソフトウェアを使う場合に必要になる場合もあります．必要になった際に調べれば良いでしょう．

2.4 コンソール入力

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

int main(){
  string s;              ← 文字列型変s宣言
  cin >> s;              ← コンソール入力からsに代入
  cout << s <<endl;      ← コンソールにsの値出力
}
```

このプログラムを走らせると，入力待ち状態になり，文字列を入力してエンターキー（以下では⏎で表します）を押すと，入力した文字列が出力されて終わります．<<を用いてコンソール出力できることを既に学びましたが，>>でコンソール入力できます．直観的には納得が行くでしょう．たとえば， abc⏎と入力すると，

```
abc
```

と表示されます．

複数の変数に入力したい場合は，繰り返すだけです．

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

int main(){
  string s;
  int n;
  cout<< "文字列，整数 => ";
  cin >> s >> n;
  cout << s << ", " << n << endl;
}
```

文字列と整数を入力し，それを表示するコードの例を書いてみました．たとえば，abc␣100 ↩ と入力すると，

```
文字列，整数 => abc 100
abc, 100
```

と表示されます．入力は，空白で区切られますが， ↩ も空白に含まれるので，abc ↩ 100 ↩ と入力しても出力は同じです．

2.5 コメント

```
/* コメントを理解するためのコード例
 * 数行にわたっても良いです
 */
#include <iostream>
using namespace std;

int main(){
  cout<<"Hello!"<<endl; // 行の終わりまで無視
}
```

コメントは，ソースコードの中で，コンパイラ，よってコンピュータに無視され，プログラムの作動に一切影響を与えない部分です．コメントは人間のためのもので，コードで何しているのかをわかりやすくするために用います．

コード内どこでも，以下のどちらもコメントとなります．

コメント

- /* と*/の間
- //から行の終わりまで

/* と*/は数行にわたることができるので，長いコメントをしたい場合に用います．//はちょっとメモ的に使う場合に便利です．

コメントのもう1つの非常に便利な用途を説明します．コーディングをしていると，この部分を書き換えたらどうなるのだろう，あるいはこの部分を無くしたらどうなるのだろう，という疑問が生じることが

あります．これは特にデバッグ[2]する際によく生じます．いくつかの指示を一時的に無効にするには，それらをコメントにしてしまえば良いのです．コメントではなくすれば元に戻せるので，安心して書き換えられます．このように，コメントにして一部の指示を無効化することを**コメントアウト**するといいます．

2.6 演算

整数の演算

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

int main(){
  int m,n;
  cout<< "m,n => ";
  cin >> m >> n;                // m,n入力
  cout << "+: " << m+n << endl; //足し算
  cout << "-: " << m-n << endl; //引き算
  cout << "*: " << m*n << endl; //掛け算
  cout << "/: " << m/n << endl; //割り算，0注意！
  cout << "%: " << m%n << endl; //余り算
}
```

+,-,*,/,%はそれぞれ足し算，引き算，掛け算，割り算，余り算を計算する**演算子**で，**算術演算子**とよびます（表2.1にまとめました）．様々な整数値を２個入力して確認してみてください．たとえば5␣3 ⏎ と入力すると以下のように表示されます．

```
m,n => 5 3
+: 8
-: 2
*: 15
/: 1
%: 2
```

演算子	操作
+	足し算
-	引き算
*	掛け算
/	割り算
%	余り算

Table 2.1: 算術演算子の種類と役割.

概ね自明でしょうが，１つ注意すべき点をあげます．C++での整数演算では結果も整数です．数学的には，割り算の答えは一般に**整数になりません**．C++の整数同士の割り算の答えは，切り捨てられた整数値となります（上の例では割り算の答が四捨五入では2ですが，1となっています）．余り算は整数固有の計算ですが，プログラミングでは大変便利な演算で，よく使います．余り算で負の整数値も扱えますが，少し複雑なのでここでは説明しません．上でnに0を入力すると，0で割ろうとしてアプリケーションが「クラッシュ」（プログラムエラー）します．これも試してみましょう（コンピューターは壊れないので心配しないで下さい）．なお，たとえばabcみたいに，整数以外の値を入力すると出力結果はどうなるか保障されていません．

[2]デバッグするとは，コードの間違いを見つけて直すことで，コード内の間違いをバグ（英語で虫の意味）と呼ぶことからきています．

実数の演算

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

int main(){
  double x,y;  //  x,yはdouble型
  cout<< "x,y => ";
  cin >> x >> y;     // x,y入力
  cout << "+: " << x+y << endl; //足し算
  cout << "-: " << x-y << endl; //引き算
  cout << "*: " << x*y << endl; //掛け算
  cout << "/: " << x/y << endl; //割り算，0注意！
}
```

直前の整数演算のコードとほぼ同じ内容の実数演算のコードです．整数の場合のコードと比較して，実質的にintをdoubleにして，余り算（実数では無意味）を取り除いただけであることに注意しましょう（double型変数には演算%は定義されていません）．慣習として整数型変数名はm,n,p,qで始め，実数型変数名はx,y,zで始める場合が多いので，慣習に従って変数名を変更しましたが，文法的にはこの変更は意味はありません．5.0␣3.0⏎ を入力すると，表示は以下のとおりです．

```
x,y => 5.0 3.0
+: 8
-: 2
*: 15
/: 1.66667
```

整数の場合と比較すると，割り算の結果が違うことがわかります．また，5.0，3.0と入力するかわりに5␣3⏎と入力しても結果は同じです．x,yが実数型変数であるので，実数として読み込まれるからです．小数点を含めた様々な値も入力できるので，いろいろ試してみてください．

C++では**累乗の算術演算子は定義されていません．**累乗を計算するには，pow関数を使います．これについては3.2.2節で説明します．

文字列の演算 — おまけ

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

int main(){
  string s1,s2;
  cout<< "s1,s2 => ";
  cin >> s1 >> s2;
  cout << "+: " << s1+s2 << endl; // 文字列の+
}
```

string型には四則演算の中で+しか定義されていませんが，便利なので説明しておきます．上のプログラムを走らせて，たとえばabc␣def⏎と入力すると，以下の出力が得られます．

```
s1,s2 => abc def
+: abcdef
```

見てわかるように，string型の+は文字列をつなげる演算です．stringについては，*，-，/，%は定義されていません．

本書では説明しませんが，これら以外にビット毎の演算も定義されています．これは二進法の計算で，ビット演算ともよびます．

まとめ：変数

- 変数は必ず型（種類）を指定して宣言してから使用．
- 典型的なデータの型の例：int，double，string

その他

- 四則演算：+-*/，整数の余り算：%
- コメント：//から行の終わりか，/*と*/の間．
- コンソール入力（cin）

コラム 1: 変数名の付け方

長い変数，関数名は煩雑で，また場所を取るので本書では，簡単なコードでは，1文字のクラス名，関数名，変数名をよく用いています．実践的にはどのような変数名を用いたら良いのでしょうか？変数名はアルファベット大文字，小文字，数字，_の組み合わせなので，かなり自由度があります．どのような変数名にすると良いか悩む場面が出てくるでしょう．

まず第一に意識すべきは，変数名はコンピュータにとっては何であろうが，同じということです．よって，変数名は人間にわかりやすいように選びます．ここでは，普通に使われる慣習の例をあげます．

- 変数名の1文字目は小文字．
- 変数名の1文字目は_にしない（1文字目は_にすると，意図せずOS，コンパイラ等で既に定義してある変数名と同じになる危険あり）．
- 意味のあるデータにはその意味がわかりやすい名前を付けて，1文字の変数名にしない．
- 言葉をつなげる場合は，大文字か_で区切る（例：`populationData`， `population_data`，`gakusekiBangou`，`gakuseki_bangou`,，等）．
- 1文字の変数名は，ループ変数，使い捨ての変数等のスコープ（9.5節で説明）が小さい変数に用いる．
- 変更しない定数を保持する変数は，大文字だけの変数名．
- クラス名の1文字目は大文字．
- クラス内の`private`メンバの名は_で終える．

クラス（第4章参照），スコープ（9.5節参照）は後に出てきます．

Chapter 3

関数

3.1 関数の構造

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

int func3x(int x){  // 整数3倍する関数定義
  return 3*x;
}

int main(){
  cout << func3x(10) << endl; // 関数をよぶ
}
```

整数を3倍する関数，`func3x`を定義し，それを使いました．プログラムを走らせた結果は以下のとおりです．

```
30
```

一般に，関数`f`は変数を与えて（**引数**とよびます），評価して値（**戻り値**とよびます）を得ます．関数は次の論理的な構造を持ちます．

関数を宣言する際は，**引数も戻り値も型を持つ**ので，それを宣言に含める必要があります．引数は複数，あるいは無くてもかまいません．

文法的には関数は以下のように宣言して定義します．

関数の宣言と定義

```
戻り値の型 関数名(引数の型 引数変数名){
  指示1;
  指示2;
  ...
  return 戻り値;
}
```

ここでは，宣言と定義を同時に行っていますが，後に分ける方法も学びます．関数をよぶと，戻り値を戻す（`return`する）わけです．

関数について，数点重要なポイントをあげます．

- 関数はコードで用いる前に宣言する必要があります（定義は後でも良いです）．

- 引数はいくつでも（複数でも，無くても）良く，型は自由です．
- 戻り値は無しの場合はvoid型の戻り値となります．
- 戻り値は1個ですが型は自由です．これは制約が厳しいように見えるかも知れませんが，後にわかるように，複数の値を戻したい場合はそれをまとめた1つの型を定義すれば良いので，制約にはなりません．
- 許される関数名は変数名と同じルールです．
- 関数の引数は関数内でしか意味を持ちません．
- 関数の引数を関数としても良いです．たとえば，関数f(x)，g(x)を定義し，複合関数f(g(x))を使うことができます．これは非常に便利です．また，f(f(x))のように自分自身をよぶこともできます（再帰的によぶといいます）．

3.2 様々な関数

3.2.1 引数や戻り値が無い関数

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

void message(){  // 引数無し，戻り値無し（void）
  cout << "Message: Hello!" << endl;
}

int main(){
  message();
}
```

初めの関数の例（3.1節参照）は戻り値と引数がともにint型の場合でした．引数や戻り値が無い関数のイメージはわきにくいかも知れませんが，上の例は引数が無く，戻り値がvoid型（戻り値データが無い）関数です．走らせると

```
Message: Hello!
```

と出力されます．この例から想像できるように，引数や戻り値が無い関数を使う状況も自然に生じます．

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

void message(string s){  // 戻り値無し（void）
  cout << "Message: " + s << endl;
}

int main(){
  string s;
  cout<<"Message => ";
  cin>>s;
  message(s);
}
```

引数は有り，戻り値が無い関数の例です．2.6節で説明した文字列の演算を用いました．たとえば，hi! ⏎ と入力すると，以下が表示されます．

```
Message => hi!
Message: hi!
```

3.2.2 様々な関数の例

定義されている関数を使う

```
#include <iostream>
#include <cmath>  // 数学関係の定義読み込む
using namespace std;

int main(){
  cout<< pow(4,3) <<endl;  // pow(x,a)はxのa乗
  cout<< pow(2,0.5) <<endl;
}
```

標準で定義されている累乗を求める関数，powを使う例です．自分で定義しなくても定義が用意されているので，それを#include <cmath>で読み込み，使います．powの引数は2つあり，pow(x,a)はx^aを与えます．プログラム出力は以下のとおりです．

```
64
1.41421
```

$4^3 = 64$，$2^{0.5} = \sqrt{2} = 1.41421$であることがわかります．pow関数は複素数の累乗計算にも用います．

複数の引数がある関数

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

void showData(string name,int age){
  cout<<"Name: "<<name<<", age: "<<age<<endl;
}

int main(){
  showData("聖徳太子",48) ;
}
```

引数が複数あり，それらの型が異なる場合の例で，プログラムを走らると出力は以下のとおりです．

```
Name: 聖徳太子, age: 48
```

3.3 デフォルト引数

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

void message(string s = "Hello!"){
  cout << "Message: " + s << endl;
}

int main(){
  message();
  message("こんにちは");
```

```
}
```

3.2.1節で説明した例を組み合わせました．出力は以下のとおりです．

```
Message: Hello!
Message: こんにちは
```

関数の引数のデフォルト値を定義しているので，関数をよぶ際にその引数の値を与えなければ，デフォルト値（上のコードでは"Hello!"が使われます．このように引数のデフォルト値を定義することを，**デフォルト引数**とよびます．

引数が複数の場合のデフォルト引数　関数の引数が複数の場合にもデフォルト引数は使えます．例をあげます．

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

int f(int m,int n,int a=1,int b=0){//デフォルト引数
  return a*m+b*n;
}
int main(){
  cout<<f(1,2)<<endl;
  cout<<f(1,2,3)<<endl;
  cout<<f(1,2,3,4)<<endl;
}
```

引数４個のうち，最後の２個のデフォルト引数を定義しています．プログラムの出力は以下のとおりです．

```
1
3
11
```

$1*1+0*2=0$，$3*1+0*2=3$，$3*1+4*2=11$が確認できます．

引数が複数の場合は，どの引数にデフォルト値を定義するかに注意が必要です．**デフォルト値は必ず右の変数から定義します．**よって，上の例では，デフォルト値を定義できる組み合わせは，(m,n,a,b)，(n,a,b)，(a,b)，(b)のどれかです．たとえば，(n,a)だけのデフォルト値は定義することはできません．

デフォルト引数

```
戻り値の型 関数名(引数の型 引数変数名,...,引数の型 引数変数名=デフォルト値,...){
  指示1;    指示2;    ...  return 戻り値;
}
```

強調しますが，引数は

(デフォルト値無しの引数，デフォルト値有りの引数)

となります．それぞれの引数は一般に複数でも０個でも良いです．デフォルト値は，コーディングを楽にするので，それで十分な場合には使えば良いでしょう．たとえば，パラメータを用いた計算におけるパラメータのデフォルト値などに使うのは自然です．

3.4 関数のオーバーロード

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

void message(){
  cout << "Message: Hello!" << endl;
}

void message(string s){
  cout << "Message: " + s << endl;
}

int main(){
  message();
  message("こんにちは");
}
```

前節のコードと作動は同じで，出力は以下のとおりです．

```
Message: Hello!
Message: こんにちは
```

当たり前過ぎて，何も新しいことは無いようにも見えるかも知れません．ここで注目して欲しいのは，同じ関数名の関数messageを2種類定義していることです．**同じ関数名の関数に2つ以上の定義をすることをオーバーロードする**といいます．

オーバーロードできる条件

引数の数や型が異なること

関数名が同じ場合は引数が異ならない限り，コンピューターはどちらの関数を使えば良いのか判別できません[1]！上の例では，片方は引数無し，もう1つは引数1個なので明確に違います．message関数はデフォルトでは，Hello!を出力し，それを変更できる，という関数の定義になっています．オーバーロードは便利な概念で，以下では必要になります．オーバーロードした関数は，元の関数と全く異なるふるまいをしても文法的には良いです．しかし，同じ関数名の関数が全く異なるふるまいをするのは間違いの元です．よって，

関数をオーバーロードする場合は，同じような作動をする関数にする．

「同じような」というのはあいまいですが，直観的に同じように見える作動をさせるということです．引数の数は同じで，型だけが違う関数オーバーローディングは間違いを起こしやすいので，よほど自信が無い限り使わないことを勧めます．

上のコードと同じ作動をするコードは，関数内でもう1つの関数を用いて次のようにも書けます．こちらの方が，無駄な繰り返しが少なくて良いです．理解してみましょう．

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

void message(string s){
  cout << "Message: " + s << endl;
}
void message(){
  message("Hello!"); //上の関数よぶ
}
int main(){
```

[1]厳密には，namespaceで区別できる場合もあります．次節参照．

```
  message();
  message("こんにちは");
}
```

上のプログラムと同じ作動は，前節でデフォルト引数を用いても得られました．どちらを使うのが良いのでしょうか？まず，オーバーロードを用いれば引数のデフォルト値を決めるだけではなく，どのような関数も書けます．そして，一般には，引数の数が異なる場合に関数のふるまいは，引数の値が異なるだけではありません．よって，同じ関数名で引数の型と数が違う関数の定義に，一般にはデフォルト引数では不十分です．

どちらでも使える場合は，どちらでも良いです．自分にとって楽，あるいは自然に感じる方を使えば良く，デフォルト引数の方が簡単な場合が多いでしょう．たとえば，上の例ではデフォルト引数の方が簡単かつ自然だと思います．

3.5 main関数

3.5.1 C++プログラムはmain関数を走らせるだけ

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

int main(){
  cout<<"Hello World!"<<endl;
}
```

これは1.2.1節の一番初めのコードです．ようやく，この全てを説明できます．実は，あらゆるC++のコードはmain関数を定義しているだけです．そして，C++のプログラムを走らせるとは，main関数を評価する，あるいは呼ぶことを意味します．ここで，戻り値の型がintなのでなぜ，値を戻していない（return指示が無い）のか気になるかも知れません．これは，main関数に限り，デフォルトで0を戻すことになっていて，return文を省略しても良いからです．つまり，上のコードは，コンピューターにとっては以下のコードと全く同じです．

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

int main(){
  cout<<"Hello World!"<<endl;
  return 0;
}
```

0の値はmain関数を呼び出しているOSに「戻して」います．return 1;（あるいは100でも）として，異なる値を戻しても文法的には良いですが，正常に作動した場合は0を戻す慣習になっていて，異常に作動した場合を検出する手段なので，特に理由が無ければ0を戻すべきです．

3.5.2 #includeとは

上のコードの他の部分も説明しましょう．Hello World!と標準出力に出力するだけでも，cout，<<とendlがどこかで定義されていなければなりません．これはiostream[2]という**ヘッダファイル**で定義されています（システム上にiostream, iostream.h あるいはiostream.hppというファイルがあるはずです）．ヘッダファイルは様々な定義をまとめたファイルで，数多くのヘッダファイルが用意されています．iostreamのヘッダファイルを読み込めという指示が#include <iostream>です．必要に応じて，他の定義も読み込みます．その例が，3.2.2節の#include <cmath>で，このヘッダファイルには数学関係の標準的な関数や値（sin，cos，pow，exp，log関数やπ，eの値等々）が含まれています．

[2]Input/output streamの略なので名前の由来はわかりやすいでしょう．

3.5.3 名前空間，namespace

namespaceは文字通り，**名前空間**を意味します．名前空間は，いくつかの関数，変数の名前を集めたものです．たとえば，cout，cin，endlの正式な名前はstd::cout，std::cin，std::endlで，std名前空間に属します．コードの2行目のusing namespace std;はstd名前空間内の関数等については，名前空間名のstdを省略しても良いという指示です．using namespace std;を使っていない次のコードを見るとわかるでしょう．作動は全く同じです．

```
#include <iostream>

int main(){
  std::cout<<"Hello World!"<<std::endl;
}
```

using namespace std;を使わないと，std名前空間のcoutという意味で， std::coutと明示的に指定する必要があります（指定しないとコンパイル時にエラーを起こします）．std::endlも同様です．一々stdを指定するのは煩雑なので，本書ではusing namespace std;を使います．

なぜ名前空間のような面倒そうなものがあるのか不思議に思うかも知れません．色々コードを書いていると，関数名が意図せず既に標準で定義されているものや，使いたい外部のライブラリのものと一致してしまう場合が少なくありません．その場合でも名前空間で区別し，どの関数を使うか指定できるのでnamespaceは重要は概念です．関数以外でも，定数，データの型等でnamespaceを使った区別が必要になる場合があります．

3.6 関数内の変数は外では使えない

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

int func3x(int x){  // 整数3倍する関数定義
  return 3*x;
}

int main(){
  x = 5; //xは宣言されていないのでコンパイルエラー
  cout << func3x(x) << endl; // 関数をよぶ
}
```

3.1節のコードにmain内にx=5;を加えた上のコードは文法エラーでコンパイルできません．それが当たり前に感じられる方には本節は必要無いかも知れません．しかし，誤解する場合もある重要な点なので，強調します．関数では引数のint xで整数型の変数xを宣言しています．この宣言は関数func3x内でしか有効ではありません．よって，関数main内では定義されていません．

このように，変数，より一般的にはデータ，にはその宣言が有効な範囲があり，それをデータの**スコープ**とよびます．スコープは{}で囲まれた範囲（**ブロック**とよびます）の1つです（スコープとブロックについては，9.5節でより詳しく扱います）．たとえば，func3x(int x)で宣言されたxは関数の{}で囲まれたブロックの中だけで使えます．また，関数のブロック内で宣言されたデータは，そのブロック内で使えます．mainのブロック内で宣言したデータはmain内の中だけで使えます．確認してみましょう．

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

int func3x(int x){  // 整数3倍する関数定義
  return 3*x;
}
```

```
int main(){
  int x = 5; //このxはmain内だけで使える
  cout << func3x(x) << endl; // 関数をよぶ
}
```

このプログラムの出力は以下のとおりです．

```
15
```

さっきのコンパイルエラーを起こしたコードとの唯一の差は`x = 5;` の前に`int` を付けただけです．これで，整数型データxを宣言して5を代入する指示になっています．ここで絶対に理解して欲しいのは，**func3xとmain内の両方にxがありますが，この2つは全く無関係の別のデータであること**です．たとえば，main内の変数名xをzに変更しても作動は変わりません．確認してみてください．

くどいかも知れませんが，次の例でさらに確認します．

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

int func3x(int x){  // 整数3倍する関数定義
  x = 3*x;
  return x;
}

int main(){
  int x = 5; //このxはmain内だけで使える
  cout << func3x(x) << endl; // 関数をよぶ
  cout << x <<endl; //xは変化していない
}
```

このプログラムの出力は以下のとおりです．

```
15
5
```

関数func3xの作動は変わりませんが，xに3倍したxを代入し，それを戻すという指示にしました．よって，func3x内ではxは変化しています．しかし，func3xをmain内でよんでも，main内のxは（全く独立な変数なので）変化していないことがわかります．

この節は「なぜ当たり前のことを説明しているだろう?」と思った方もいるかも知れません．そう思った方は安心して良いです．難しく感じた方は，重要な点なのでしっかり理解して下さい．

3.7 関数：応用

```
#include <iostream>
#include <cmath> // sqrt等数学関係定義読み込み
using namespace std;
// 2次方程式の解
double rootDiscriminant(double a,double b,double c){
  return sqrt(b*b-4*a*c);
}
double sol1(double a,double b,double c){
  return (-b+rootDiscriminant(a,b,c))/2.0/a;
}
double sol2(double a,double b,double c){
```

```
  return (-b-rootDiscriminant(a,b,c))/2.0/a;
}

int main(){
  cout<<"Solve equation, a x^2+b x+c=0, a,b,c => ";
  double a,b,c; // 使う直前に宣言
  cin>>a>>b>>c;
  cout<< sol1(a,b,c) <<"\t"<< sol2(a,b,c) <<endl;
}
```

2次方程式$ax^2+bx+c=0$の解$(-b\pm\sqrt{b^2-4ac})/2a$を計算するコードを上に示しました．出力例を2つ下に示します．

```
Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => 1 -2 1
1 1
```

$x^2-2x+1=0$の解は$x=1$の重根です．

```
Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => 3 1 -1
0.434259 -0.767592
```

$3x^2-x+1=0$の解の$x=(-1\pm\sqrt{13})/6$（有効数字6桁）です．

このプログラムには少し問題があります．たとえば`0 1 1`⏎と入力した場合のように，`a`が0となる場合は0で割ることになります．また，`1 1 1`⏎と入力した場合は$x^2+x+1=0$が実数解を持たないので，`rootDiscriminant`の戻り値が`double`なので取り扱えません．入力してみるとわかりますが，`nan`（not a number，「数ではない」）や`inf`（infinity，無限大）といった出力をします．本来は，入力値に応じて，作動を変更すべきです．これをするには，道具がまだ足りないので，5.9節でまた扱います．

まとめ：関数

- 関数は戻り値，引数の型を指定して定義.
- 関数では`return`を用いてデータを戻す.
- 引数は型も数も複数でも，空でも良い.
- 戻り値の型は1つだが，`void`（空）型でも良い.
- コード：`main`関数の定義.
- プログラム作動：`main`関数の評価.

コラム 2: デバッギング

コードを書いて，コンパイルして走らせたら，一発で意図通りに作動する—これがプログラミングの理想ですが，現実にはよほど単純なコードでも無い限り，そういうことはまずありません．コードには，多くの場合間違い（バグ）があり，それらを取り除く必要があります．この作業をデバッギングとよびます．

コードを書いたらコンパイルしなければ走りませんが，文法間違いでコンパイルしない場合があります．この場合は，コンパイラのエラーメッセージを読んでみましょう．慣れていないと，メッセージが何を行っているのかさっぱりわからないかも知れませんが，ヒントにはなります．どの行に問題があるかを指摘してくれるだけでも，大いに助かります．エラーメッセージを見るときには，必ず初めの方からチェックしていきます．初めのバグが後のエラーの起因かも知れないからです．初めからバグを取り除けば後のエラーは自然と消えることも多いです．警告とエラーのメッセージがあり，エラーは必ず対応が必要です．警告はコンパイルはしても，間違った作動を引き起こすコードの場合もあるので，警告にも目を通しましょう．

プログラミングの経験が浅いうちは，コードがコンパイルするまでが大変で，コンパイルしたら，万々歳でほとんど仕事が終わったと思うかも知れません．意外かも知れませんが，一般にコンパイルするコードのバグの方が，見つけて直すのが困難です．コンパイルするコードのエラーは，意図通りにプログラムが作動しないということです．コンピュータに（意図せず）間違った指示をしているのです．

敢えて自分の苦労を増やすように聞こえるかも知れませんが，できる限りコンパイル時にバグを見つけられるようにします．変数のスコープを小さくする方針（9.5節参照）もその例です．コンパイル時のエラーは増えるかも知れませんが，コンパイルした後の手間が減ります．その意味では，コンパイル時のメッセージもできる限り詳細に出すようにすることを勧めます．全ての警告に対応する必要は無いかも知れません．ただ，どういう問題を指摘されているかに目を通しておくと，間違いに早く気付きやすいです．

コンパイルしたコードは，バグがあるかわからない段階でも，典型的な場合で必ず作動を確認すべきです．作動が意図通りでない場合は，デバッギングが必要です．作動が正しくない場合は，まずコードを調べます．それで問題がわかり，直して正しく作動する場合はそれで良いでしょう．それでもわからない場合は，データを出力するコード（たとえば `cout<<`データ···）を途中に加えて，作動させ，途中の状態を確認します．最終的に誤動作していることがわかっても，どの部分で間違いが起きているか絞り込む必要があるからです．途中出力をして，どの段階まで，正しい作動をしていて，どこから間違った作動をしているか特定します．どこに間違いがあるかわかれば，大分楽です．このデバッギングの方法は，単純かつ原始的で，新たなソフトウェアもいりませんが強力です．

デバッギングには，`gdb`等のソース・レベルデバッガが便利なソフトウェアツールです．ソース・レベルデバッガを使うと，コードの途中で作動を一時停止（ブレークポイントを入れるといいいます）し，また再開できます．その際，その時点でのデータの値を表示することもできます．こういった操作は，コードを一切変更しないでできます．さらに，データの変化を見ながらコードを 1 行ずつ（あるいは任意の行数ずつ）実行したりすることもできます．使うためには，少し慣れが必要ですが，本格的にコーディングするのであれば使ってみることを勧めます．

Chapter 4

クラスとオブジェクト

4.1　クラス，オブジェクトとは

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

class A{  //クラスAの宣言始まり
public:
 void print(){ cout<<"I am A"<<endl;} //メンバ関数
};        //クラスAの定義終わり

int main(){
  A a; // クラスAのオブジェクトa
  a.print(); //メンバ関数を使う
}
```

このプログラムの出力は以下のとおりです．

```
I am A
```

これまではint，double，stringなど，標準的に用意されているデータの型を用いてきました．この節では，**クラス**（class）を用い，自分でデータの型を定義します．クラスの中にはデータと関数を含められます．クラスを宣言，定義するには，メンバとなる（中に含める）データと関数を宣言し，定義します．クラス内のデータを**メンバデータ**，関数を**メンバ関数**とよびます．メンバデータの型には自分で定義したクラスも使えます．

基本的な概念を上の例で見てみましょう．Aという名前のクラスを定義しました．Aにはメンバ関数printだけが定義されています．main内のA a;で，A型のデータaを宣言し，aのメモリも確保しています．このように，クラス Aのデータを作ることを**実体化**とよびます．aをAの**オブジェクト**とよびます．

クラスは型の定義（設計図），オブジェクトはクラスを実体化したもの（実体）

です．a.print()はオブジェクトaの関数printをよびだしています．オブジェクト内のメンバを指定する際に .（ピリオド）を使います．実体を**インスタンス**，実体化を**インスタンス化**ともよびます．

クラスの宣言のしかたは以下のとおりです．

クラス宣言

```
class クラス名{
public:
  メンバデータ，メンバ関数の宣言，定義
private:
  メンバデータ，メンバ関数の宣言，定義
```

```
};
```

クラス外からのメンバのよびだし

オブジェクト名.メンバ名

クラス宣言では，最後に；を付けることを忘れないでください．忘れると文法エラーになります．

クラス宣言で，public，privateはアクセス制御を指定しています．アクセス制御とは，どこからメンバデータ，メンバ関数にアクセスできる（よびだせる）か指定することです．アクセス制御には以下の3種類あります．

アクセス制御

- public：クラス外からよびだせる．
- private：クラス内（とそのfriendクラス内）からしかよびだせない．
- protected：クラス内，そのfriendクラス内，とそれらを継承したクラス内からしかよびだせない．

friend，継承については第8章で学びます．C++では，メンバはデフォルトではprivateになります．たとえば，上の例でpublic:の行を無くしてみて下さい．クラス外のmainからメンバ関数をよぼうとしているので，文法エラーでコンパイルできません．どれがpublic，privateなのかすぐにわかるように，明示的にpublic，privateの順序で宣言することを勧めます．上の例にはprivate部分がありませんが，privateについては次の例で実感できるでしょう．

オブジェクトは，多機能の自動販売機みたいなものです．様々な材料（データ）を中に保持していて，様々な指示（メンバ関数）をすると，指定したものを出（出力）します．コーヒーのクリームの量を指定するなど，細い指示もできる場合もあります．メンバ関数を使ってオブジェクトに指示を送るので，メンバ関数やそれを使うことをメッセージ，メソッド，やメッセージングとよぶこともあります．

クラスとオブジェクトについて数点付け加えます．

1. メンバデータ，関数は何種類あっても良い．
2. privateにできるものはする．

2点目は文法的には従う必要はありませんが，C++のように，クラス，オブジェクトを使ったプログラミングの基本的な考え方です．プログラミングを楽にするには，**整理**する必要があります．関係するものを1まとめに整理するのがクラスです．クラスはデータとそのデータの処理法を一緒にまとめられるので便利です．クラス内へのアクセスは最小限にし，それぞれのクラスの独立性を高くした方がより整理されていて使いやすくなります．privateにすれば，外からアクセスできないので，その部分の独立性を担保できます．

身近な例で考えてみましょう．様々な書類が多くあったら，それを全部一緒にしたら使うのは難しいので，それをフォルダ（クラス）に分けたりするでしょう．そして，その書類の説明，データの使い方（メンバ関数）とかのメモもフォルダに入れると便利です．フォルダに分けたとしても，一般にはフォルダを越えて関係，関連があるでしょう．それでも，できるだけ内容がフォルダ毎に独立している方が使いやすいはずです．

上のコードと全く同じ作動をするコードです．

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

class A{  //クラスAの宣言始まり
public:
  void print(){ cout<<s<<endl;}  //メンバ関数定義
private:
  string s = "I am A"; //private変数
```

```
};        //クラスAの定義終わり
int main(){
  A a; // クラスAのオブジェクトa
  a.print(); //メンバ関数を使う
}
```

ほとんど違いはありませんが，privateなstring sの宣言と定義がクラス内でされています．宣言，定義の仕方は今までと同じです．print関数はクラス内で定義された関数なので，private変数sを呼び出せます．メンバ関数から同じクラスのメンバデータを使うには，変数名をそのまま用いれば良いことに注意して下さい．以下で例が出てきますが，同じクラスのメンバ関数を呼び出す場合も，関数名をそのまま用います．

4.2 宣言と定義

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

class A{  //クラスAの宣言始まり
public:
  void print();  //メンバ関数宣言
};        //クラスAの定義終わり

void A::print(){ cout<<"I am A"<<endl;}  //メンバ関数定義

int main(){
  A a; // クラスAのオブジェクトa
  a.print(); //メンバ関数を使う
}
```

上のコードは作動は4.1節のコードと全く同じですが，メンバ関数printの宣言と定義をはっきり分けました．見てわかるように，宣言では関数の関数名，引数，戻り値の型を指定します．定義では，実際にその関数の作動を指定します．クラスのメンバ関数のは，**クラス外では**そのクラスのメンバ関数であることがわかるように，::を用い，A::print()はクラスAのprint()メンバ関数という意味となります．クラスの宣言内では，そのあいまいさは無いのでA::は付けません．データについても同様です．

メンバ関数宣言はクラス宣言内，定義はクラス宣言外とする場合は以下のとおりです．

<u>クラス宣言内のメンバ関数宣言</u>

戻り値型 メンバ関数名(引数型 引数名);

引数は複数でも良いです．**最後の ; を忘れないでください．** メンバ関数はクラス宣言の{}内で宣言はしなければなりません．

<u>クラス宣言外でのメンバ関数定義</u>

クラス名::メンバ関数名(引数型 引数名){ 指示1; 指示2; ...}

定義は4.1節のようにクラス宣言の{}内でしても，上のコードのように，外でも良いです．メンバ関数が複数行にわたる場合は，コードがわかりやすいように，クラス宣言の{}の外で定義するのが普通です．クラス宣言内はメンバの宣言だけで，定義は必ず外という流儀もあります．クラス宣言**内**でメンバ関数を定義することを，**インライン定義**，宣言**外**で定義することを**非インライン定義**とよぶこともあります．

宣言と定義を別にすることは，メンバ関数に限らず一般の関数で行なえます．3.1節の関数の例と同じ作動をするコードを以下に示します．

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

int func3x(int); //関数宣言

int main(){
  cout << func3x(10) << endl; //関数よぶ
}

int func3x(int x){  //関数定義
  return 3*x;
}
```

関数の宣言と定義を分けました．関数の宣言は，使う前に必要ですが，定義は後でも良いです．この点がわかるように，定義は後に持しました．関数の宣言では，引数，戻り値の型を指定します．関数の宣言があれば，関数の定義は別のファイルで行っても良いです．定義と宣言を分けるのは，関数のコードが長い場合に別のファイルにして整理する，等の理由があります．

4.3 コンストラクタ

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

class A{  //クラスAの宣言始まり
public:
  A(){ n = 0 ; }  // コンストラクタ
  void setN(int n0){ n = n0; }  //セッター
  int getN(){ return n; }  //ゲッター
private:
  int n;
};        //クラスAの定義終わり

int main(){
  A a; // クラスAのオブジェクトa
  cout << a.getN() <<endl;
  a.setN(100);
  cout << a.getN() <<endl;
}
```

このプログラムの出力は以下のとおりです．

```
0
100
```

上のコードでは，クラスAの宣言にメンバデータ（整数型 n）と，メンバ関数を含めています．新しいのは宣言にクラス名と同じ関数名のメンバ関数，A()があることです．このようなクラス名を関数名に持つ関数は**コンストラクタ**です．コンストラクタ宣言には戻り値の型指定はありません．コンストラクタにはreturn文もありません．

<u>デフォルトコンストラクタ宣言と定義</u>

```
class クラス名{
  クラス名(){ 指示1; 指示2; ...}
```

```
};
```

コンストラクタはクラスを実体化する際によばれます．よって，そこで，オブジェクトの初期化（必要に応じてメンバデータの初期化，メモリの確保，等）を行います．この例のように，引数無しのコンストラクタを**デフォルトコンストラクタ**とよびます[1]．4.1節の例では，コード内でコンストラクタは定義しませんでした．その場合は，最低限のデフォルトコンストラクタがよばれ，メンバデータ，メンバ関数が使えるようになります．メンバデータの初期化等の操作はされません．

publicとprivate　ここでpublicとprivateの意味の確認をしておきましょう．main関数内で，

```
cout << A.a <<endl;
```

とメンバデータを直接出力しようとすると，privateメンバにクラス外から直接アクセスしようとしているので，文法間違いになり，コンパイルできません．なお，メンバデータをpublicに指定すると，メンバデータを直接出力できます．

セッターとゲッター　setN，getNは，セッター，ゲッターとよばれ，メンバデータを変更，出力する関数です．メンバデータはクラス内だけからしかアクセスできないprivateなメンバにする場合が多いです．その場合，クラス外からアクセスするためにセッター，ゲッターを用います．セッターを用いてメンバデータを変更し，ゲッターで出力します．メンバデータをprivateにすると，メンバデータを変更する際にセッターを通すので，範囲を制御などを通じてデータの整合性を保ちやすいこと，メンバデータの形式を必要に応じて変更しても，クラス内だけの変更で済む，などの利点があります．セッター，ゲッターはよく使われますが，常に必要なわけではありません．クラス外からアクセスしないデータには無意味です．また，public指定してクラス外から直接アクセスしても問題無い場合もあります．

4.4 コンストラクタのオーバーロード

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

class A{  //クラスAの定義始まり
public:
  A(){ n = 100 ; }  //デフォルトコンストラクタ
  A(int n0){ setN(n0); } //引数有りコンストラクタ
  void setN(int n0){ n = n0; }  //セッター
  int getN(){ return n; }  //ゲッター
private:
  int n;
};        //クラスAの定義終わり

int main(){
  A a0; //デフォルトコンストラクタよぶ
  cout << a0.getN() <<endl;
  A a1(200); //引数有りコンストラクタよぶ
  cout << a1.getN() <<endl;
}
```

コンストラクタも関数の1種なので，3.4節で説明したとおり，オーバーロードできます．これを示したのが上のコード例で，実行すると次の出力を得ます．

[1]言葉の問題ですが，明示的にコンストラクタを定義しない場合のデフォルトのコンストラクタは，デフォルトコンストラクタの特殊な場合です．

```
100
200
```

オーバーロードをできる条件は，引数の数や型が異なることで，上の例では引数の数が0，1個で異なります．クラスを実体化する際に，引数無しでよべばデフォルトコンストラクタ（引数無しコンストラクタ）がよばれ，引数を含めてよべば，引数があるコンストラクタがよばれます．

上ではコンストラクタの定義と宣言の両方をクラス宣言内（インライン定義）で行いました．

<u>クラス宣言内でのコンストラクタ宣言と定義</u>

```
class クラス名{
  クラス名(引数){ 指示1; 指示2; ...}
};
```

コンストラクタもメンバ関数なのでクラス宣言内でのコンストラクタの宣言は必要ですが，定義はクラス宣言外でも良いです（4.2節参照）．

<u>クラス宣言内でのコンストラクタ宣言</u>

```
class クラス名{
  クラス名(引数);
};
```

<u>クラス宣言外でのコンストラクタ定義</u>

```
  クラス名::クラス名(引数){ 指示1; 指示2; ...}
```

コンストラクタを呼び出して，オブジェクトを実体化する方法は以下のとおりです．引数は一般に複数です．

<u>オブジェクト宣言と実体化</u>

```
クラス名 オブジェクト名(コンストラクタ引数の値);
```

これは上のコードで使った方法で，オブジェクト名の定義と宣言を分けることもできます．

<u>オブジェクト宣言と実体化</u>

```
クラス名 オブジェクト名;
オブジェクト名 = クラス名(コンストラクタ引数の値);
```

コンストラクタについて重要な注意点をあげます．

I **引数があるコンストラクタを定義した場合は，デフォルトコンストラクタは明示的に定義しない限り無くなる．** 上のコードで，`A(){ ···}`の定義を無くすと，A a0;の宣言は文法エラーになります．確認してみましょう．引数有りのコンストラクタを定義していなければ，明示的に定義しなくてもデフォルトコンストラクタ（引数無し）が用意されていた点に注意しましょう．

II **コンストラクタでデフォルト引数を使える．** コンストラクタも関数の1種なので，デフォルト引数（3.3節参照）が，同じ文法に従って使えます．便利なので，必要に応じて使いましょう．

III **コンストラクタから同じクラスのコンストラクタをよばない．** 上のコードで，デフォルトコンストラクタを`A(){ A(100); }`と定義すれば経済的でスマートだと思うかも知れません．これは意図通りには作動しません．間違いやすい点なので注意しましょう．このように書くと，`A a0;`と指示した場合に`n`を100としたオブジェクトが作られますが，それは使われずに，`a0`は`n`を初期化されていない`A`のオブジェクトとなります．難しい事を考えなくて良いので，コンストラクタから同じクラスのコンストラクタをよばないようにしましょう．

コンストラクタのオーバーロードを扱いましたが，他のメンバ関数も関数の 1 種なのでオーバーロードできます．

第1章で，変数を宣言，定義するのに，たとえば，intであれば

```
int n = 100;
```

でも

```
int n(100);
```

でも良いと書きました．後者の書き方は，int型のコンストラクタによる初期化，とイメージするとわかりやすいでしょう．

4.5 コンストラクタにおけるデータ初期化

クラス宣言内でのコンストラクタ宣言

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

class Student{
public:
  Student(string n,int y): name(n),year(y){} //コンストラクタ
  void print(){ cout<<name<<": "<<year<<"年生"<<endl;}
private:
  string name;
  int year;
};
int main(){
  Student s("C++太郎",1);
  s.print();
}
```

プログラムの出力は以下のとおりです．

```
C++太郎: 1年生
```

新しいのは，コンストラクタで値をメンバデータに代入する文法です．ここでは，コンストラクタで，メンバデータ初期化しか行っていませんが，もちろん，{}内であらゆる操作の指示が行えます．

この例のコンストラクタは，

```
Student(string n,int y){ name = n; year = y; }
```

と同じ意味です．こう書いても良いです．初めのコード例の書き方はメンバデータ初期化していることがわかりやすく，簡潔なので，こちらもよく使われます．

コンストラクタでのメンバデータ初期化

```
クラス名(引数1,引数2,...) : メンバデータ1(引数1),メンバデータ2(引数2),...{
指示;...
}
```

4.6 const修飾子

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

int main(){
  const int n = 100; //const
  cout << n <<endl;
  n = 200; // 文法間違い
}
```

const[2] 修飾子はデータの型に加えると，データは改変できないデータとなります．上のコードはコンパイルできません．constであるはずのnを変更しようとしているからです．constは，変更しない定数などを宣言して定義する際によく使います．

constはメンバ関数で使う場合には異なる側面もあるので，これを説明します．

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

class A{  //クラスAの宣言始まり
public:
  A(){ n = 100 ; }  //デフォルトコンストラクタ
  int getN() const { return n; }  //const
private:
  int n;
};        //クラスAの定義終わり

int main(){
  A a0; //デフォルトコンストラクタよぶ
  cout << a0.getN() <<endl;
}
```

4.3節のコードのゲッター定義にconst修飾子を含めました．メンバ関数名の後にconst修飾子を加えると，**メンバデータを一切変更しないメンバ関数**という意味になります．ゲッターはクラスのメンバデータを変更しないことを文法的に保障しています．

const修飾子を使う場合は，関数などをよぶ際によばれた関数が，const宣言で不定性を保障しておく必要があることに注意が必要です．const修飾子は，変更できないことを宣言するだけなので，文法的に必要な場合はありません．変更できるものを変更しなくても良いからです．ただし，const修飾子に反して変更しようとすると，コンパイルエラーになります．こう考えると，面倒なだけで，どのような状況で使うのだろう，と疑問に思うかも知れません．意外かも知れませんが，原則は，**constは使える箇所では常に使う**，です．変更すべきでないものは，変更しようとした場合にコンパイルできないので，走らせる前に間違いを捕まえられるからです．このように，できる限りコンパイル時にプログラミングの間違いを見つけようとするのは**防衛的プログラミング**の考え方の一例です．初めは面倒に感じるかも知れませんが，コンパイル時にチェックを厳しくして，実行する前にできるだけ間違いを減らしておくことは，特に大掛かりなプログラムを書く場合には時間節約につながります．また，constを用いると，変化しないデータであることがわかるので，コンパイラが最適化をしやすくなる利点もあります．

4.7 デストラクタ

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

class A{
```

[2] constはconstant（一定，不変，定数等の意味）の略です．

```
public:
  A() { cout<<"Constructed"<<endl; }//コンストラクタ定義
  ~A(){ cout<<"Destructed"<<endl; } //デストラクタ定義
};
int main(){
  A a;
}
```

コンストラクタはオブジェクト生成時によばれますが，オブジェクト破棄時によばれるのが**デストラクタ**です．上のコードを走らせると，次の出力が得られます．

```
Constructed
Destructed
```

オブジェクトを`A a;`で宣言しただけですが，それにより，コンストラクタが走り，オブジェクトが実体化されます．そして，プログラム終了時に破棄されるので，デストラクタが走ります．デストラクタは以下のように宣言します．クラス宣言内で定義する場合は，他のメンバ関数同様に**クラス名**`::`は使いません．

デストラクタ定義

```
クラス名::~クラス名(){  指示; }
```

オブジェクトはスコープ（3.6節,9.5節参照）から外れれば破棄されます（プログラム終了時までには必ずスコープから外れます）．デストラクタはその際に自動的によばれるメンバ関数で，引数は取りません．よって，オーバーロードもできません．

デストラクタは，コンストラクタ同様に，明示的に定義しなくても標準的に用意されるデストラクタが走り，オブジェクトのメモリ等を解放します．デストラクタは，「後始末」をすべき場合に用います．典型的な使用例は，オブジェクト生成時に明示的にメモリを確保した場合は，それを明示的に解放する必要があり（6.5節参照），デストラクタに解放させます．これをしないと，**メモリーリーク**とよばれるメモリが開放されない厄介なプログラム間違いを引き起こします．

4.8 なぜクラスを使うのか — プログラミング法の進化

なぜ，クラスを使うのでしょうか？コンピュータを動かすには機械語の実行形式のプログラムがあれば良いだけです．極端な話をすれば，機械語で書けば何でも書けるわけで，高度な言語すら必要ありません．機械語で書くのは人間には難しいので，プログラミング言語を用います．それでも，たとえばC言語を使えば何でも書けます．なぜクラスを導入したC++言語を用いるのでしょうか？

クラスを使えば，プログラミングが楽になるから使います．どのようにしたら，楽に正確にプログラミングできるのでしょうか？それは，整理しやすくすることです．プログラミング手法の発展を見返すと，構造化プログラミング，オブジェクト指向，といった新たな考え方の導入は，より楽な整理法方の導入でした．

クラスを用いることにより，関連のあるデータとその処理法をひとまとめに整理できます．そして，それぞれのクラスの出入口を管理し，できるだけ独立させれば，管理しやすくなり，コードの再利用もしやすくなります．たとえば，`private`なものはいくら書き換えても，クラス内だけの影響に抑えられます．必要な機能を加える際は，メンバ関数を加えれば良いだけです．プログラミングが楽で楽しくなります．もう1つクラスの理解が必要な大きな理由は，比較的新しい既成のツール類を用いるためには，クラスを使いこなす必要がある場合が多いからです．特に，現代的な手法でGUIアプリケーションを作るためには，クラスを使いこなすことは必須です．

クラスも使いようによっては，もちろん便利にも不便にもなります．整理しやすくするために使うので，各クラスに，比較的独立して，再利用しやすいように切り分けて，まとめる必要があります．複雑な課題であればあるほど，クラスのデザインが大事です．

4.9 struct

クラスを宣言，定義するのにclassを使っていますが，classを使う代わりに，structを用いることもできます．classとstructの違いは，メンバデータ，メンバ関数がclassではデフォルトでprivate指定，structではpublic指定であることです．それ以外に違いは全くありません．よって，public，privateを明示している場合は，コードでclassをstructに入れ替えてもコンパイルしたプログラムは全く同じ作動をします．本書では，常にclassを用います．classを用いる方が標準的で，他のプログラミング言語とも共通です．

なぜclassがあるのに，structがあるのかという疑問があるかも知れません．実は，C++言語の前身のC言語に構造体の型としてstructがありました．大雑把にはこれはpublic，privateのアクセス制御ができない，メンバデータだけを含むメンバ関数が使えないクラスです．C++言語はC言語の文法を受け継いでおり，C言語にはアクセス制御が無いので，デフォルトがpublicのstructが残ったのです．

4.10 クラスとオブジェクト：応用

```
#include <iostream>
#include <cmath> //sqrtに必要
using namespace std;

class Solve2{
public:
  Solve2(double a,double b,double c): a_(a),b_(b),c_(c){}
  Solve2(double b,double c): a_(1),b_(b),c_(c){}
  double rootDiscriminant() const { return sqrt(b_*b_-4*a_*c_);   }
  void printSol() const { cout<<"Solutions: "<<sol1_()<<"\t"<<sol2_()<<endl; }
private:
  double a_,b_,c_;
  double sol1_() const{  return (-b_+rootDiscriminant())/2.0/a_; }
  double sol2_() const{  return (-b_-rootDiscriminant())/2.0/a_; }
};
int main(){
  cout<<"Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => ";
  double a,b,c;
  cin>>a>>b>>c;
  Solve2 solve(a,b,c);
  solve.printSol();
}
```

3.7節の2次方程式の解を出力するコードをクラスを用いて書き換えた例です．たとえば4␣-3␣-1 ⏎ と入力すると，以下の出力を得ます．

```
Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => 4 -3 -1
Solutions: 1 -0.25
```

$4x^2-3x-1=0$の解が$x=1,-1/4$であることがわかります．使ってはいませんが，デフォルトでは２次の項が係数が1となるようにコンストラクタをオーバーロードしています（１つ目の変数なので，デフォルト引数は使えません）．privateメンバ名は_で終える流儀を使っています．3.7節のコード同様に，$a=0$の場合や実数解が無い場合には対応していません．

クラスとオブジェクト

- クラスは設計図，それを実体化したものがオブジェクト．

- クラスはメンバデータ，メンバ関数を含む.
- クラスはコンストラクタ，デストラクタを含む．明示的に定義しなくても最低限のものは用意されている.
- `public/private`:クラス外からアクセスできる/できない.

<u>`const`</u>

`const`：変化しない宣言．原則として使えるときには使う.

Chapter 5

流れの制御

C++プログラムを走らせると，`main`プログラムの指示を上から実行します．このプログラムの単純な流れを，より多様な作動をさせるためのコーディングの仕方を学びます．

5.1 if

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

int main(){
  double x;
  cout<<"real number => ";
  cin>>x;
  if( x>0 ){ // 正であればブロック実行
    cout<<">0" <<endl;
  }
}
```

上のプログラムを走らせると，もし正（> 0）の数を入力すると，`>0`と出力されます．「もしそうであれば」という条件は`if`で指定され，コードの構造は以下のとおりです．

if文

```
if(条件){
 指示;
}
```

`if`の条件が満たされる場合に限り，{}内の指示（一般には複数）を実行します．

ブロック　{}内の指示のまとまりを**ブロック**とよびます．よって，`if`文は，`if`の条件が満たされれば，その次のブロックを実行するということになります．実は，`if`のブロックは指示が1つだけの場合は{}を省略できます．しかし，これは編集中に`if`ブロック内の指示を増やそうとした際などに間違いやすいので，指示が1つでも{}でくくることを勧めます．

5.2 条件

5.1節では正（> 0）の条件判定をしました．C++では以下の条件判定の演算子を使えます．条件判定した結果の値は**真**（true），か**偽**（false）です．0以外の整数は真，0は偽と判定されます．

演算子	意味	例，説明
`==`	等しい	`a == b`
`!=`	等しくない	`a != b`
`<`	未満	`a < b`
`<=`	以下	`a <= b`
`>`	より大きい	`a > b`
`>=`	以上	`a >= b`
`!`	not（否定）	`!(a==b)`は`a!=b`と同値
`&&`	and（かつ）	`(a==b)&&(b==c)`はa=b=cの意味
`||`	or（または）	`(a==b)||(b==c)`は，a=bかb=cの意味

その性質上，`==`，`!=`，`<=`，`<`，`>=`，`>`は**比較演算子**，`!`，`&&`，`||`は**論理演算子**とよばれます．`A || B`は，`A`，`B`どちらかが真の場合，`A`，`B`両方とも真の場合のどちらの場合にも真です（包含的論理和）.

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

int main(){
  cout<<"integer => ";
  int n;
  cin >> n;
  if ( n==1 ){// 1に等しい
    cout<<"if ( n==1 )"<<endl;
  }
  if( n!=1 ){ // 1 ではない
    cout<<"if ( n!=1 )"<<endl;
  }
  if( n>0 && n<10 ){//0と10の間
    cout<<"if( n>0 && n<10 )"<<endl;
  }
  if( n<=0 || n>=10 ){//0以下か10以上
    cout<<"if( n<=0 || n>=10 )"<<endl;
  }
  if( !(n>1) ){ //1より大きくはない，n>1は()必要
    cout<<"if( !(n>1) )"<<endl;
  }
  if( n ){  //整数で真偽
    cout<<"if ( n )"<<endl;
  }
}
```

条件が真である場合にその条件を出力して確認するコードを上に示しました．いろいろな整数値を入力して理解を確認してみてください．`0`,`-1`を入力した例を示します.

```
integer => 0
if ( n!=1 )
if( n<=0 || n>=10 )
if( !(n>1) )
```

```
integer => -1
if ( n!=1 )
if( n<=0 || n>=10 )
if( !(n>1) )
```

```
if ( n )
```

上の例で，0は偽，-1は（0以外なので）真と判定されていることが確認できます．

5.2.1 条件判定で注意すべき点

注意すべき点をあげます．

- 等しいの条件は==であり，=ではない（=は代入）．
- かつ（&&），または（||）は&,|を２個続ける（&，|の1個の演算子は違う意味を持つ）．
- 数式でも，論理式でもどの順序で評価されるか悩むようなら( )で囲む．
- 条件判定で0以外は真．

上の条件に関する重要な点をもう少し詳しく解説します．

等しい条件の判定は ==

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

int main(){
  int x = 3;
  if( x = 5 ){ // 代入！条件判定ではない
    cout << "x = 5" <<endl;
  }
}
```

上のプログラムを実行すると，

```
x = 5
```

と出力されます．x = 3のはずなのに，なぜだろうと思うかも知れません．上のコードは文法的には正しいですが，上のifは，「もしもxに5を代入し，それが正常に作動すれば次のブロックを実行する」という意味になります．代入は正常に作動するので，ブロックが実行され，x = 5と出力されるわけです．条件判定は==であるのにうっかり=（代入）を使うとこういうことが起きます．文法的に正しいのに，作動が間違っている場合は問題を見つけにくいのでこういったミスは危険です．通常コンパイラが警告してくれるので，警告にも注意しましょう．if(x == 5)と書けば，（多分意図どおりに）xが5であるかの判定になります．

なぜ，条件判定の中で指示を実行できる仕様なのか疑問に思うかも知れませんが，実はこれは便利です．ファイルを開く，データを確保する，等の指示をし，それが正常にできたら次の操作をする，などといった流れをif条件内に指示を含めて処理することがよくあります．このような使い方の例も後に説明します．

演算子の優先順位 たとえば，

```
x0 > x1 && x1 > x2 || x2 > x3
```

を考えてみましょう．意味は次の２つのどちらでしょうか？

```
(x0 > x1 && x1 > x2) || x2 > x3
x0 > x1 && (x1 > x2 || x2 > x3)
```

１番目が正しいですが，()を使えば論理が見てすぐわかります．解釈に迷うくらいであれば，()を使って間違いにくくしましょう．どの演算子が先に評価されるかの順序を，演算子の優先順位とよびます．優先順位は>,<の方が&&より高く，&&の方が||より高いわけです．算数で$3+5\times5$と書くと，$3+(5\times5)$を意味するのと考え方は同じです．上の例をよりはっきりと書くのであれば，

++　--　!
*　/　%
+　-
<<　>>
<　<=　>　>=
==　!=
&&
||
=　+=　-=　*=　/=　%=

Table 5.1: 演算子の優先順位：上にあるほど先に評価される．()内では，その中での優先順位．

```
((x0 > x1) && (x1 > x2)) || (x2 > x3)
```

となります．>,<の方が&&,||より優先順位が高いのは直感的に明らかに感じますが，これは慣れの問題があります．とにかく，迷うようなら，()でくくって，優先順位を明示しましょう．この考え方は，普通の演算でも論理式でも共通します． 表5.1に主な演算子の優先順位をまとめておきました（まだ登場していない演算子も含めてあります）．

0以外は真

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

int main(){
  double x;
  cout<<"=> ";
  cin>>x;
  if( x ){//真
    cout<<"true" <<endl;
  }
}
```

真偽値はブール代数の値（true，false）で表されます．真偽を数字で表す場合には，通常は整数を用いますが，上のコードを実行して様々な数値を入力すればわかるように，実数でも0以外は真と解釈されます．

5.3　if，else if，と else

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

int main(){
  int n;
  cout<<"integer => ";
  cin>>n;
  if( n==0 ){//もし
    cout<<"0" <<endl;
  }
  else if( n>0 ){ //そうでなくもし>0
    cout<<">0" <<endl;
  }
```

```
  else { // それ以外
    cout<<"<0"<<endl;
  }
}
```

if構文として，もしAならば，そうではなくBであれば，それ以外は，と言った構文の例が上のコードです．if（もしも），else if（そうではなく，もしも），else（それ以外の場合は），という構造です．英語での意味どおりなので，わかりやすいでしょう．下にまとめます．

```
if，else if，else

if(条件1){
 指示;
}
else if(条件2){
 指示;
}
else if(条件3){
 指示;
}
...
else{
 指示;
}
```

if，else if，elseいずれについても，条件が満たされれば次に続くブロックの指示を実行します．なお，else ifは0個でも複数個あっても，良いです．elseは無くても良いです．よって，一般には，次のいずれかの構造となります．

- if
- if，else if
- if，else
- if，else if，else

必ずifで始まり，else if，elseはそれより前で一番近くにあるifに付随します．以下で気をつけるべき点を数点あげます．

ifのネスティング（入れ子）

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

void ifNest(int n){
  cout<<"Input: " << n <<"\t"; //\tはタブ
  if( n>0 ){
    if( n>1 ){
      cout<<"n>1\t" ;
    }
    cout<<"n>0" ;
  }
  cout << endl;
}
```

```
int main(){
  ifNest(-10);
  ifNest(1);
  ifNest(10);
}
```

if，else if，elseの中でif，else if，elseをネスティングする（「入れ子」構造にする）ことも可能です．上のコードはその簡単な関数を使った例です．n>0の場合の中で，n>1の場合は更に指示があるという構造です．上の例ではifしかネスティングに含めていませんが，else if，elseもさらに含めることももちろんできます．\tはタブを意味し，出力を見やすくするために含めました．プログラム出力は，以下のとおりです．

```
Input: -10
Input: 1 n>0
Input: 10 n>1 n>0
```

本書では，if，else if，else等の実行ブロックを必ず{}でくくっていますが，ネスティングする際は，この習慣を特に強く勧めます．そうしないと，どのifに付随しているのかの混乱を引き起こしかねません．

else if，elseはそれ以前の条件以外

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

int main(){
  int n;
  cout<<"integer => ";
  cin>>n;
  if( n==0 ){//もし
    cout<<"0" <<endl;
  }
  else if( n==1 ){ // それ以外でもしも1
    cout<<"1" <<endl;
  }
  else if( n>0 ){ //それ以外でもしも>0
    cout<<">0" <<endl;
  }
  else { // それ以外
    cout<<"<0"<<endl;
  }
}
```

上のコードは以前のコードとほとんど同じですが，else if (n>0)の前にelse if (n==1)を入れました．1を入力すると，次の出力がされます．

```
integer => 1
1
```

n=1はn>0も満たしていますが，else if (n>0)の「それ以外」は上にある条件すべて以外であり，n==1の条件を満たしたので，実行されません．論理的に考えれば納得行くと思いますが，注意しましょう．

5.4 while

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

int main(){
  int j = 0; //初期条件
  while( j<10 ){ //j<10の間
    cout<<j<<" "; //jの値出力
    ++j;
  }
  cout << endl;
}
```

プログラムの出力は以下のとおりです．

```
0 1 2 3 4 5 6 7 8 9
```

whileは「...の間」の英語の意味通り，条件が満たされている間はその直後のブロックを実行し続けます．++jはjを1増やすという指示で，以下で詳しく説明します．whileは次のように使います．

```
while

while(条件){
  指示;
}
```

上のコード例のように，whileは，ループ（繰り返し）をするために使います．繰り返しはコンピューターの得意技で，便利であり，プログラムミングの基本です．

5.4.1 +=,-=,*=,/=,++,--

+=

```
  j += k;
```

は

```
  j = j+k;
```

の省略形です．kは変数でも定数でも良いです．単なる省略なので，j = j+k;が文法的に正しければ，j += k;も使えます．

-=,*=,/= -,*,/も全く同様です．+.-,*,/が定義されている変数であれば，+=,-=,*=,/=は使えます．

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

int main(){
  double x = 1.1;
  x += x;
  cout << x << endl;
  x += 0.1;
  cout << x << endl;
  x /= 2;
  cout << x << endl;
  string s = "abc";
```

```
  s += "def";
  cout << s <<endl;
}
```

上のコードでいくつかの例を試しています．出力は以下のとおりです．

```
2.2
2.3
1.15
abcdef
```

理解を確認して，自分でいろいろ試してみてください．

++ ++は変数に1を加える演算子です．よって，

```
  ++j;
```

はjに1を加える指示で

```
  j++;
```

と

```
  j = j+1;
```

と

```
  j += 1;
```

と同値な指示です．ループでは，繰り返し1回毎に1増える整数（ループカウンタ）を使うことが多く，1増やす指示を頻繁に使うので，++のように便利な演算子が用意されています．++j;とj++;は同じ指示です．このように，++を使う際は指示はそれだけにしましょう．本書ではそうします．そうしないと，わかりにくいコードになりがちです．[1]

++は1を加える演算子なので，1を加えられる変数であれば使えるので，実数型変数にも使えます．ただ，整数型変数のように1つ大きくする，1先に進める，等が自然な場合に用いることがほとんどです．

-- --は++の逆で，1減らす演算子です．++の場合と同様に，--j; はj--;，j = j-1;j -= 1;と同じ指示になります．

++，--は演算子を及ぼす変数との間に空白をはさんでも使えます．たとえば++j;は++ j;とも書けます．この点は他の演算子と同様ですが，++，--は変数との間に空白をはさまずに使うのが普通です．

5.5 break

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

int main(){
  int j = 0;
  while( 1 ){ //while(true)も同じ
    if( j>= 10 ){
      break;  //ループ脱出
    }
    cout<<j<<" ";
```

[1] ++jとj++は変数jをよぶ前に1を加えるか，後に加えるかの差です．(++j)+(++j);といった指示も可能で，この場合はj++と++jのどちらを使うかの差があります．ただ，わかると思いますが，読み取りにくいコードになりがちなので，++j;のように1を加える指示だけにしましょう．

```
    ++j;
  }
  cout<<endl;
}
```

5.4節のwhileループは，条件を満たしている間実行されますが，breakを用いて，ループ途中で脱出できます．その例が上のコードで，プログラムの作動は5.4節のコードと同じです．breakはループを脱出する指示です．このコードではwhileの条件は常に真なので，途中でループを脱出しない限り，無限ループになります．この例では，10以上になればループ脱出します．ループを使う際には，意図せず無限ループを作ることは少なくないので，気を付ける必要があります．

breakは，異常事態の場合に脱出する，解を探している際に見つけたらループ止める，無限ループ防止の安全のために入れる，等様々な使い道があります．

5.6 for

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

int main(){
  for(int j=0 ; j<10 ; ++j){
    cout<<j<<" ";
  }
  cout << endl;
}
```

forループは**一番よく使うループ構造**です．習得しましょう．上のプログラムも5.4節と同じ作動をします．forの構文は次のようになります．

```
for

for(初期条件 ; 継続条件 ; 繰り返す前に最後に行う指示 ){
  指示;
}
```

すぐ気づくように，forを使うとループが簡潔にまとまります．詳しく見ていきましょう．forのあとの()内には;に区切られた以下の3つの箇所があります．

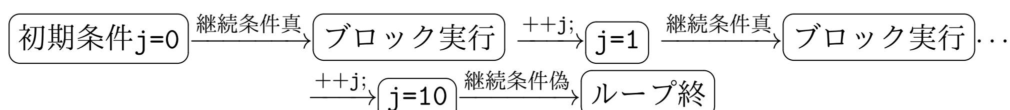
1. **初期条件**：ループを始める前の指示です．上の例のようにループカウンタ初期化などが典型的な指示です（ループ内処理を 1 回こなすごとに 1 増える変数をループカウンタとよびます）．

2. **継続条件**：この条件が満たされている間は直後ブロックを繰り返し実行します．

3. **繰り返す前に最後に行う指示**：ブロックを実行した後，繰り返すかの継続条件を判定する前にこの指示を実行します．

この3つの箇所は，いずれも空でも構いません．空の指示は何もしない指示で，空の継続条件は**真**になります．

上のコードを理解しましょう．初期条件はint j=0;で，整数型変数jを宣言し，値0で定義します．継続条件はj<10 で j が10未満であれば繰り返します（5.4節のwhileの条件に対応します）．ブロックの指示を実行し，継続条件を判定する前に++j;，ループカウンタjを1増やすことを行います．ループの作動をまとめると次のようになります．



このまとめにも表したように，forループの継続条件はブロック実行する前に必ず確認します．よって，初期条件が継続条件を満たしていなければ，ブロックが1回も実行されないという事態が生じます．実行されないループを作る人はいないだろうと思うかも知れませんが，動的に（プログラム内の状態に依存して）初期条件を設定している場合には，間違いでなくても生じる場合があります．

5.6.1 ループ内変数

上のコード例では

```
for(int j=0 ; ... ; ... )
```

のようにforの中でint j=0;と変数jを定義しました．このように定義すると，jはループの**ブロック内だけ**で定義されています．つまり，jのスコープはforブロックです．forブロック外では定義されていないので使えません．

```
int j;
for(j=0 ; ... ; ... )
```

と書いても，上のプログラムの作動は変わりませんが，jはforのブロック外でも使えるようになります．どちらの方が良いのでしょうか？より広い範囲で使い回せた方が便利に思うかも知れません．

ループ内の変数でループ内に限定できるもの（ループカウンタ等）はループ内に限定するのが大原則です．より一般的には，あらゆるデータに関して，定義されている範囲を必要最小限に留めるのが原則です．その方が間違いが起きにくいからです．これも防衛的プログラミングの考え方の1つです．スコープについては9.5節でより詳しく説明します．

5.6.2 forはwhileで書ける

5.4節，5.6節のコード例が同じ作動をすることから想像できるように，for文はwhile文で書けます．5.6節のfor文の構造は以下のように書けます．[2]

```
初期条件 ;
while(継続条件){
  指示;
  繰り返す前に最後に行う指示;
}
```

5.6.3 whileはforで書ける

逆に，whileもforで書けます．5.4節のwhile構文は

```
for( ; 条件 ; ){
  指示;
}
```

と同じです．このように，forで()内の3箇所の一部を空にする場合は多いです．

for(;;){···}は，while(1){···}と同じで，このままではずっと繰り返し続けるループになります（forの継続条件が空の場合は真と解釈される）．意外かも知れませんが，このような使い方も珍しくありません．この場合は無限ループにしないために，適切な条件下でbreak;指示をします．

5.7 do while

[2]厳密に同じにするためには，全体を{}でくくり，初期条件で使われる変数のスコープを同じにします．

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

int main(){
  int n;
  do{
    cout<<"n (positive) = ? => ";
    cin>>n;
  } while (n<0); //n<0の間繰り返す
  cout << n <<endl;
}
```

for，whileループに比べると使う機会は少ないですが，もう1つのループ構造にdo whileループがあります．do whileループは，whileの条件が満たされている間，ブロックを実行しますが，**条件の判定はブロックの後に行う**のがポイントです．上の例は，0以上の入力があるまで入力し続けさせるコードです．たとえば，−1 ⏎ 1 ⏎ と入力した場合の表示を示します．

```
n (positive) = ? => -1
n (positive) = ? => 1
1
```

do while構文は以下のとおりです．

```
do while

do{
  指示;
} while(条件);
```

while(条件)後の;**を忘れない**ように注意しましょう．

for，whileループがあるのに，do whileループを使う理由はわかりにくいかも知れません．do whileループの特徴は，条件判定が指示ブロックの後であるため，必ず1回はブロックを実行することです．これは，上の例のように，ある条件を満たす入力を求める場合などに便利です．上の例をfor，whileを使っても書くことはできますが，その場合は，breakを使わない限り，ループに入る前に1回入力を求め，そして，条件を満たさない場合にループに入るという構造になり，指示をループ前と中で繰り返す無駄が生じます．

5.8 switch

```
#include <iostream>
using namespace std;
void switchTest(int n){
  switch(n){
  case 0:
    cout<< "0"<<endl;
    break; //break無いと次も実行
  case 1:
    cout<< "1"<<endl;
    cout<<"etc."<<endl; //指示複数OK
    break; //break無いと次も実行
  default: //上の場合以外
    cout<< "Other"<<endl;
  }
}
```

```
int main(){
  switchTest(0);
  cout<<"----"<<endl;
  switchTest(1);
  cout<<"----"<<endl;
  switchTest(-1);
}
```

switchは変数の値によって場合分けする構文です．上のプログラムの出力は以下のとおりです．

```
0
----
1
etc.
----
Other
```

switch(n)ではnの値がcaseの後の値に一致する場合にその後の指示を実行します．気をつけなければいけないのは，その後の指示をずっと実行するので，break;を使わないと意図しない他の指示も実行してしまう場合があることです．break;をコメントアウトして試してみましょう．caseで扱っていない値の場合は最後のdefaultの指示を実行します．defaultは必要無ければ無くても良いです．switchの構文は次のとおりです．

switch

```
switch(変数){
case 変数の値:
  指示;
  break;
....
default:
   指示;
}
```

上の例では，整数型変数を用いましたが，他のデータ型の変数もswitchに使えます．

気付いたと思いますが，switchはif，else if，elseを用いて書き直せます．たとえば，上の例のswitchTest関数は次のように書き換えられます．

```
void switchTest(int n){
  if( n ==  0 ){
    cout<< "0"<<endl;
  }
  else if( n == 1 ){
    cout<< "1"<<endl;
    cout<<"etc."<<endl; //指示複数OK
  }
  else{ //上の場合以外
    cout<< "Other"<<endl;
  }
}
```

switchとif，else if，elseのどちらを使った方が良いのでしょうか？ルールはありません．両方使える場合には，自分にわかりやすい，自然に感じる方を使えば良いです．if，else if，elseの方が応用範囲が広く，使われる頻度は圧倒的に高いです．ただ，選択肢を選ぶ場合などはswitchの方が自然に感じる場合もあるかも知れません．

5.8.1 乱数

乱数と擬似乱数

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

int main(){
  cout<<rand()<<endl;  //乱数，常に同じ！
}
```

乱数を出力するコードの例を上にあげました．実行するとたとえば次の出力が得られます．

```
1804289383
```

コンピュータで乱数を使う状況は多くあります．たとえば，ゲームでは予測できない結果が求められる場合があります．このようにランダムな結果を得たい場合には**乱数**を用います．コンピュータで発生させる乱数は**擬似乱数**で，厳密には乱数ではありません．長い周期を持ち，要素間の関係が無い数列から1つずつ数字を得ます[3]．有限な資源を持つ決定論的なハードウェアを用いる以上しょうがありません．randは0からRAND_MAXで定義される最大値までの整数を出力します．

#include <cstdlib>でC言語のstdlib.hを読み込み，rand()関数によって擬似乱数が得られます．しかし，上のコードのままでは，コメントにあるように，何回走らせても**同じ数字が出力されます**．試してみてください．これでは，乱数として機能しませんが，コンピュータは決定論的なので止むを得ません．

乱数の種　解決策としては，**擬似乱数の種**を異なるものにします．種には現時刻を使う場合が多いです．擬似乱数の種は，擬似乱数の初期条件を設定するので，時刻が異なれば違う乱数が出力されます（擬似乱数の数列でスタートする箇所が異なります）．

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

int main(){
  srand(time(NULL));//乱数初期化
  cout<<rand()<<endl;//乱数発生
}
```

srand(time(NULL));の指示を加えるだけで，走らせる毎に異なる数が得られ，実質的に乱数として使えます．試してみてください．time(NULL)は現在時刻を秒単位で表します[4]．srandは乱数の種（seed）を決める関数です．秒単位なので，あまり速く繰り返して走らせると，同じ数が出ることがあります．

ただ，テストする際等には，再現性が必要な場合もあり，その場合は，たとえばsrand(0);のように特定の数の種を使います．0でなくて，他の整数でももちろん良いです．

乱数の応用

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

int main(){
  srand(time(NULL));
```

[3]厳密には関係が無いわけではなく，関係がわかりにくく，次の数が予測するのが困難（現実的には無理）な数列です．

[4]1970年1月1日 0:00 (UTC/GMT)からの秒数．

```
  for(int j=0 ; j<10 ; ++j){ //サイコロ10回
    cout<<rand()%6+1<<" ";
  }
}
```

サイコロを10回振るコード例をあげました．出力はたとえば以下のとおりです．

```
4 2 3 2 2 4 4 4 1 5
```

プログラムを走らせるたびに出力は異なります．擬似乱数を使う場合は，初期化（種の設定）はプログラムを走らせる際に1回行えば十分です．上の例でも，初めにsrandを用いて初期化し，その後はrandで擬似乱数を発生させているだけです．これは論理的に納得がいくでしょう．逆に，何回も初期化しても，ランダム性が上がるどころか，むしろ一般には落ちます．上の例では，forブロック内にsrand(time(NULL));入れて，毎回初期化すると，大半が同じ数になり，乱数としての役割を果たしません（timeが秒単位なので同じ種になるからです）．

（擬似）乱数とその使い方について説明しました．randは標準的に使えますが，乱数としてより高い無作為性を求めるのであれば，boost等の高品質のライブラリを用いることが考えられます[5]．また，乱数の種として時刻が秒単位では荒すぎる場合にはミリ，マイクロ秒単位の時刻を用いることもできます．時刻以外にも，キーボードでキー入力間隔も種に用いるなど，擬似乱数の無作為性を高める方法は他にもあります．

5.9 流れの制御：応用

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

class Solve2{
public:
  Solve2(double a,double b,double c): a_(a),b_(b),c_(c){}
  Solve2(double b,double c): a_(1),b_(b),c_(c){}
  double rootDiscriminant() const { return sqrt(discriminant_());   }
  void printSol() const;
private:
  double a_,b_,c_;
  double sol1_() const{  return (-b_+rootDiscriminant())/2.0/a_; }
  double sol2_() const{  return (-b_-rootDiscriminant())/2.0/a_; }
  double discriminant_() const{ return b_*b_-4*a_*c_;   }
};

void Solve2::printSol() const {
  if( a_==0 ){
    cout<<"2次の係数が0!"<<endl;
  }
  else if( discriminant_()<0 ){
    cout<<"実数解無し"<<endl;
  }
  else{
    cout<<"解: "<<sol1_()<<"\t"<<sol2_()<<endl;
  }
}
```

[5] boostライブラリは標準ではないですが，https://www.boost.org/から無料でダウンロードして使えます．

```
int main(){
  cout<<"Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => ";
  double a,b,c;
  cin>>a>>b>>c;
  Solve2 slv(a,b,c);
  slv.printSol();
}
```

4.10節の２次方程式を解くコードを，実数解が無い場合と，２次の係数が0の場合に対応させたものです．`if`，`else`等の制御が行えるようになり，条件分けができるようになりました．以下にいくつか実行例を示します．
実数解がある場合の例：

```
Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => 3 -2 -1
解: 1   -0.333333
```

実数解が無い場合の例：

```
Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => 1 1 1
実数解無し
```

2次項の係数が0の場合の例：

```
Solve a quadratic equation, a x^2+b x+c=0, a,b,c  => 0 1 1
2次の係数が0!
```

まとめ：制御

- `if` + (`else if`) + (`else`)
- 比較演算子：`==`，`!=`，`<=`，`<`，`>=`，`>`
- 論理演算子：`!`，`&&`，`||`
- `while` ループ：`while(条件) 指示;`
- `break`
- `for` ループ：`for(初期条件; 継続する条件; 一回毎の指示)`{···}
- `switch`： `break;`を忘れない．

乱数

- `srand(time(NULL))`：擬似乱数の初期化
- `rand()`：擬似乱数発生

Chapter 6

配列，ポインタとvector

本章では，配列，ポインタとvectorについて説明します．ポインタは難しいと言われることも多いですが，論理的に考えれば決して難しくありません．

6.1 配列

6.1.1 配列の使い方の基本

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

int main(){
  const int n(20);
  int a[n];  //配列の宣言，定義
  for(int j=0 ; j<n ; ++j){
    a[j] = j;
  }
  for(int j=0 ; j<n ; ++j){
    cout<<a[j]<<" ";
  }
  cout<<endl;
}
```

プログラムの出力は以下のとおりです．

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

コードでは，

```
  int a[n];  //配列の宣言，定義
```

で，整数の**配列a**を宣言し，さらに整数n個分のメモリを確保しています（コード例ではnは20）．**配列**とは，**同じ型のデータ複数個をまとめて取り扱うデータ**です（図6.1参照）．型は`int`，`double`，`string`といった標準で定義されているものでも，自分で定義したクラスでも良いです．各データを**要素**とよび，**配列名a**とするとそれぞれの要素を`a[0]`，`a[1]`$\cdots$でアクセスします．`[]`内にある$0,1,\cdots$の配列の要素を指定する数を**インデックス**（あるいは**添字**）とよびます．**インデックスは0から始まる**ので，配列の大きさが`n`であれば，最後の要素は`a[n-1]`です．

<u>配列の宣言，定義</u>

```
  データ型 配列名[要素数];
```

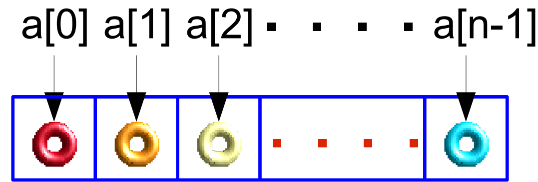


Figure 6.1: 配列の概念図.

配列の各要素

```
配列名[0]，配列名[1]， ...，  配列名[要素数-1]
```

後に説明しますが，自分でコーディングをする際は，配列を使うよりもvectorの方が便利で，vectorを使うことに実質的な不利益はありません．よって，特に理由が無い限りvectorを使うことを勧めます．しかし，vectorを使う場合にも，配列の概念の理解は不可欠です．さらに，他人の書いたソフトウェアを活用する際，配列の理解が必要な場合は多いです．

配列とループカウンタの番号の振り方 第5章や本節で様々なループを扱いましたが，その際ループカウンタ次のように0から始めました．

```
for(int j=0 ; j<n ; ++j) //普通のループカウンタの使い方
```

たとえば，n回繰り返すだけであれば，次のループカウンタの使い方もできます．

```
for(int j=0 ; j<=n-1 ; ++j) //あまり使わない
```

```
for(int j=1 ; j<=n ; ++j) //あまり使わない
```

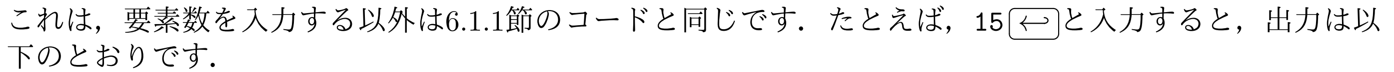
文法的には，上の3つどれを使おうが，n回繰り返すループになります（中でbreakしない限り）．通常，C++では1番目の用法のように0から番号を始め，<nの条件を用いてn-1まで数える方が普通です．この1つの大きな理由は配列等はインデックスが0からスタートするので，こちらの方が自然だからです．そして，要素n個であれば，この数字が明示されるようにした方が間違いをしにくいです．よって，本節初めのコードの番号の振り方を通常用います．ただ，コードをわかりやすくするための振り方なので，他の振り方(1,あるいは他の数から始める，など）の方がより自然であれば，それを採用すると良いでしょう．

6.2 配列と動的メモリ確保

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

int main(){
  cout<<"n => ";
  int n;
  cin>>n; //nはコンパイル時未定
  int a[n];
  for(int j=0 ; j<n ; ++j){
    a[j] = j;
  }
  for(int j=0 ; j<n ; ++j){
    cout<<a[j]<<" ";
```

```
  }
  cout<<endl;
}
```



これは，要素数を入力する以外は6.1.1節のコードと同じです．たとえば，15⏎と入力すると，出力は以下のとおりです．

```
n => 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

微々たる差に思うかも知れませんが，ここには重要な差があります．6.1.1節のプログラムでは，配列の要素数はコンパイル時に既に決まっていました．それに対し，今回は実行時に決まります．つまり，コンパイル時には決まっていなくて，プログラムを実行し，要素数を入力した後に決まります．よって，配列のメモリ確保は動的（プログラム始まりではなく，実行中に）に確保されます．これを**動的メモリ確保**とよび，これができるのはC++の重要な特徴です．プログラムを実行し始め，入力，あるいは，プログラムの計算結果等をもとに，必要に応じてメモリを確保することができるのです．よって，とりあえず十分に大きい配列を作っておこう，といったコーディングをする必要は無いと同時に，すべきではありません．

6.3 ポインタ

6.3.1 ポインタとは

この節では**ポインタ**について説明します．初め，配列と関係無いように見えるかも知れませんが，すぐ後に明らかになるように配列とは本質的な関係があります．ポインタは「難しい」と言われることも多いですが，理屈を理解すれば決して難しくはありません．落ちついて考えましょう．

C++で扱うデータには`int`，`double`，`string`のように標準的な型のものから，自分で設計したクラスのオブジェクトまで様々なものがあります．いずれの場合にも，そのデータはコンピューターのメモリ上に格納されていて，そのデータの格納場所は**アドレス（番地）**が指定されています．この単純な事実を意識すればポインタは理解できます．

たとえば，`int`型変数`n = 100`である場合を図6.2に図示しました． **ポインタとはデータのアドレスを**

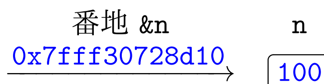


Figure 6.2: `n = 100`のデータ．値（100）とその（メモリの）アドレス（`0x7fff30728d10`）が存在する．

保持する変数です．今まではデータを保持する変数を扱っていましたが，アドレス自体もデータなので，これを保持する変数も考えることができます．それがポインタです．

変数`n`のポインタを`np`とすると，2つには関係があるので，一方からもう片方を求めることができます．ポインタのデータ（アドレス）は変数`n`から`&n`で得られます．変数からそのアドレスを求める操作を**参照**（あるいは**レファレンス**）とよびます．逆に，データはポインタ`np`の前に`*`を付けた`*np`で得られます．ポインタからデータを得る操作を**デファレンス**とよびます．この関係を図6.3にまとめました．

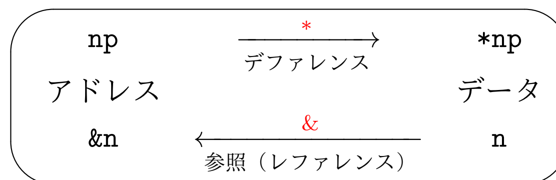


Figure 6.3: ポインタ`np`と変数`n`の関係．

上の内容をプログラムで確認します．

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

int main(){
  int n  = 100;
  int *np; //宣言：整数変数へのポインタ
  np = &n; //npをnへのポインタとする
  cout<<n<<"\t"<<*np<<endl; //データ同じ
  n = -1;  //データ変更
  cout<<n<<"\t"<<*np<<endl; //変更してもデータ同じ
  cout<<np<<"\t"<<&n<<endl; //アドレスも同じ
}
```

このプログラムの出力例は以下のとおりです．

```
100 100
-1 -1
0x7fff21b7fb6c 0x7fff21b7fb6c
```

まず，整数型変数nを宣言し，100を代入します．ポインタnpを宣言し，nへのポインタとします．すると，直接はポインタの指すデータ*npを初期化していませんが，nのデータと同じです．これは，nのデータを-1に変更しても同じです．なお，npのデータはnのアドレス（0x7fff21b7fb6c，16進法表示)であることもわかります．アドレスは一般に，OSに依存し，さらにプログラムを稼働する毎に異なります．

念のために次のポインタを使わないコードと比較して理解を確認します．

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

int main(){
  int n  = 100;
  int m;
  m = n; //
  cout<<n<<"\t"<<m<<endl; //データ同じ
  n = -1;  //データ変更
  cout<<n<<"\t"<<m<<endl; //変更するとデータ違う
  cout<<&n<<"\t"<<&m<<endl; //アドレス違う
}
```

この場合の出力例は以下のとおりです．

```
100 100
-1 100
0x7fffae454660 0x7fffae454664
```

m = n;では，変数mにnの値を代入するので，値は一致します．mはnの**コピー**です．ただ，m,nは**別の変数**なので，nの値を変更してもmの値は**変更されません．**別の変数であることは，アドレスが異なることから明らかです（出力最後の行)．ポインタとアドレスについて説明してきたことが理解できていれば，上の2つのプログラムのふるまいは当たり前に感じるはずです．そう感じない場合は，落ち着いて読みなおして理解しましょう．

C++言語は使いやすいようC言語から進化していて，C言語に比較してにポインタをあからさまに扱う必要がある場合は少ないです．しかし，C++言語でコーディングするにはポインタの理解は不可欠です．さらに，以下で説明するように，GUIを含むアプリケーションの現代的なプログラミングや，ポリモーフィズムを使うためには必要です．

6.4 オブジェクトへのポインタ

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

class A{
public:
  void print(){ cout <<"I am A" <<endl; }
};

int main(){
  A aObj; //オブジェクト実体化
  A *pa; //オブジェクトへのポインタ
  pa = &aObj; //aObjを指す
  aObj.print(); //.でメンバ呼び出し
  (*pa).print(); // *paはaObjと同じ
  pa->print(); //->でメンバ呼び出し
  cout<<&aObj<<"\t"<<pa<<endl; //アドレス同じ
}
```

このプログラムの出力は以下のとおりです．

```
I am A
I am A
I am A
0x7ffcf6e5093f 0x7ffcf6e5093f
```

一般のデータを保持する変数のポインタを定義することができるので，当然オブジェクトへのポインタも使えます．上にそのコード例を示しました．6.3.1節の単純な型，`int`のコード例とほとんど同じです．オブジェクトにはメンバがあるので，その取り扱い方が新しいだけです．既に学んだように，オブジェクトからメンバ関数，データを呼び出すには，`.`を用いて<u>`オブジェクト名.メンバ名`</u>を使います．<u>`*ポインタ`</u>がオブジェクトであることを思い出せば，<u>`(*ポインタ名).メンバ名`</u>でも呼び出せます（`()`で<u>`*ポインタ名`</u>を囲むことは必要です）．ただ，ポインタからメンバを呼び出す場面は多いので，

`ポインタ->メンバ`：オブジェクトのメンバの呼び出し方

と直接呼び出す方法が用意されています．通常はこれを用います．

6.5 new/delete

6.5.1 new，deleteの基本

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

int main(){
  double *xp; //double型データへのポインタ
  xp = new double;
  *xp = 3.14159;
  cout<<*xp<<endl;
  delete xp;
}
```

前節までの例では，ポインタを使う場合に実際のデータを保持する変数を定義し，そのポインタを定義していました．たとえば，前節の例ではaObjとpaを定義しています．これは二度手間です．データを保持する変数だけを定義する例を今まで扱ってきましたが，逆にポインタだけを定義することもできます．そのコード例が上の例です．プログラムの出力は以下のとおりです．

```
3.14159
```

ポインタを定義し，そのデータを初期化し，それをcoutに出力する，という基本的構造を持ったコードです．データを保持する変数は宣言していません．注意しなければならないのは，**ポインタ宣言しても，そのポインタが指すデータのメモリは確保されていない**，ということです．確保されるのはアドレスを保持するためのポインタ自体のメモリだけです．newはポインタの指す先のデータを実体化します．実体化はメモリ確保も含みます．データは一般のオブジェクトで良いです．メモリをnewを用いて確保した場合は，必要無くなったらdeleteを用いて，解放すべきです．上の例ではこのnew，deleteの指示が含まれています．newを用いたら，deleteを忘れないようにしましょう．

new，deleteの使い方は以下のとおりです．

new，delete

```
データ型 *ポインタ名;
ポインタ名 = new データ型;
(ポインタ使って処理)
delete ポインタ名;
```

上で説明した内容は，合理的ですが，ポインタを使う場合に実体化を忘れてプログラムがエラーを起こす場合が多いので注意しましょう．たとえば，上の例は以下のようにもっと簡潔に書ける！と思うかも知れませんが，それは間違いです．

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

int main(){
  double *xp;
  *xp = 3.14159;  //メモリ確保されていない！
  cout<<*xp<<endl;
}
```

このコードはコンパイルはします（コンパイラが親切であれば警告を出すはずです）．しかし，メモリエラーを起こしてクラッシュします．このように，コンパイルはできて間違いがあるコードの間違い探しは一般に難しいです．そういった典型例がメモリ関係の間違いです．

6.5.2 new/delete を用いたオブジェクト実体化/破棄

デフォルトコンストラクタ オブジェクトにnew，deleteを用いるのも，デフォルトコンストラクタ（引数が無いコンストラクタ）を用いる場合は，次例のように全く同じです．

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

class A{
public:
  A(){ cout<<"A constructed"<<endl;}//コンストラクタ
  ~A(){ cout<<"A destructed"<<endl; } //デストラクタ
  void print(){ cout <<"I am A" <<endl; }
};

int main(){
```

```
  A *pa; // pointer
  cout<<"Start"<<endl;
  pa = new A; //コンストラクタ稼働
  pa->print();
  delete pa; //デストラクタ稼働
  cout<<"The end"<<endl;
}
```

6.4節の例を応用しました．このプログラムの出力は以下のとおりです．

```
Start
A constructed
I am A
A destructed
The end
```

どこで，オブジェクトが実体化され，廃棄されるかを意識するために，コンストラクタとデストラクタを明示的に定義しました．newでコンストラクタが走り，実体化され，deleteでデストラクタが走り，廃棄されていることが出力から確認できます．

引数があるコンストラクタ newを用いて引数があるコンストラクタを呼んでオブジェクト実体化する例が次のコードです．

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

class Student{
public:
  Student(string s,int n): name_(s),id_(n){}
  void print() const { cout<<"Name: "<<name_<<", id: "<<id_<<endl; }
private:
  string name_;
  int id_;
};

int main(){
  Student *s1; //ポインタ
  s1 = new Student("Alice",1001); //コンストラクタ引数
  s1->print();
}
```

プログラムを走らせると次の出力を得ます．

```
Name: Alice, id: 1001
```

考え方は同じですが，コンストラクタに引数を渡す方法を知っておく必要があります（4.4節参照）．引数無しと有りの場合をまとめておきます．

newを用いたオブジェクト実体化（コンストラクタ引数無し）

```
  オブジェクトへのポインタ名 = new クラス名;
```

newを用いたオブジェクト実体化（コンストラクタ引数有り）

```
オブジェクトへのポインタ名 = new クラス名(引数);
```

引数は一般に複数です．デフォルトコンストラクタをよぶ場合には，new クラス名();でも実体化できます．上のコード例では，s1の指すオブジェクトはmainプログラム終了とともに，デストラクタにより廃棄されるので，deleteは省きました．

6.6 配列の正体

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

int main(){
  const int n = 20;
  int a[n]; //配列宣言
  for(int j=0 ; j<n ; ++j){
    a[j] = j;
  }
  cout<< a<<"\t"<<&(a[0])<<endl; //aはa[0]へのポインタ
  cout<< *a<<"\t"<<a[0]<<endl;
  cout<< a+10<<"\t"<<&(a[10])<<endl;//ポインタ演算
  cout<< *(a+10)<<"\t"<<a[10]<<endl;
}
```

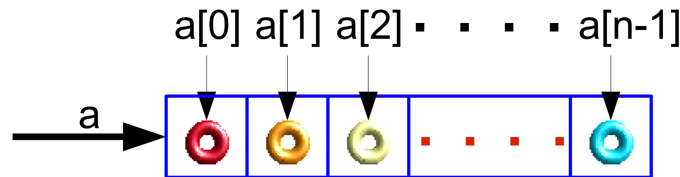


Figure 6.4: 配列とポインタの関係.

配列a[0],a[1],...を考えましょう．aの中身は何でしょうか？aは，図6.4に示したように，実は配列の先頭の要素，a[0]へのポインタです．a[j]はaからj個先の要素の変数です．これを確認しているのが上のコードです．プログラムの出力は以下のとおりです．

```
0x7ffca4f53a60 0x7ffca4f53a60
0 0
0x7ffca4f53a88 0x7ffca4f53a88
10 10
```

aはa[0]のアドレス&(a[0])と同じで，データも同じです（出力1,2行目）．aからj個先の要素へのポインタはa+jと表せます．よって，a+10はa[10]のアドレス，&(a[10])，と一致し，データも一致することが確認できます．a[j]と*(a+j)は同じで，状況，好みで使いやすい方を使えば良いです．前者の方が直観的にわかりやすい場合が多いように私は感じます．

ポインタ演算 上のa+10のように，ポインタを一定個数の要素分だけ動かす操作は**ポインタ演算**の例です．引き算，-，も使えます．ただし，メモリ確保している範囲外にアクセスしようとすると，メモリのエラーを起こすので注意しましょう．たとえば，上の例でa+1000にアクセスしようとするとコンパイルはしますが，ランタイムのエラーを起こすか，意図していない数字を出力します．この問題はポインタに限ったことではなく，a[1000]とアクセスしようとする場合にも同様な注意が必要です．

6.7 配列の new，delete

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

int main(){
  int *a; //配列は単にポインタ
  cout<<"n => ";
  int n;
  cin>>n; //nはコンパイル時未定
  a = new int [n]; //配列のメモリ確保
  for(int j=0 ; j<n ; ++j){
    a[j] = j;
  }
  for(int j=0 ; j<n ; ++j){
    cout<<a[j]<<" ";
  }
  cout<<endl;
  delete a; //new使ったらdelete忘れない
}
```

上のコードは，6.2節のプログラムと同じ作動をする，new，deleteを用いて書いたプログラムです．配列は単に先頭要素へのポインタなので，要素のデータ型（上ではint）のポインタとして宣言できます．ポインタの指すメモリ領域確保には，既に学んだようにnewを用います．唯一新しいのは，要素n個分のメモリを確保する以下の指示です．

newを用いた配列の動的メモリ確保

```
  new 要素データ型 [要素数];
```

newでメモリ確保した場合は，deleteは忘れないようにしましょう．

6.1.1節，6.2節のコードに比べて面倒に見えるかも知れません．まず，6.1.1節のコードでは，配列の大きさはコンパイル時に決まっていました．それに対し，6.2節のコードは動的に配列の大きさを決められる利点がありました．上のコードは1つ利点があります．それは，動的に大きさを伸縮自在に変更できることです．newで配列のメモリを確保します．大きさを変更する場合には，データを廃棄することにはなりますが，deleteしてから，またnewで違う大きさの配列にできます．ただし，次に説明するvectorを使えばさらに柔軟性があります．

6.8 vector

```
#include <iostream>
#include <vector> //vector使う場合必要
using namespace std;

int main(){
  cout<<"n => ";
  int n;
  cin>>n;
  vector<int> a(n);  // []ではなく()
  for(int j=0 ; j<a.size() ; ++j){//a.size()は要素数
    a[j] = j;
  }
```

```
  for(int j=0 ; j<a.size() ; ++j){
    cout<<a[j]<<" ";
  }
  cout<<endl;
}
```

vectorは配列のパワーアップ版で，クラスです．上のコードは6.2節，6.7節のコードと同じ作動をします．vectorの使い方を見ていきましょう．

```
  vector<int> a(n);
```

でint型の要素n個のvector，a，の宣言と実体化をしています．vectorに続いて，要素の型は<>で囲みます．これは，テンプレートの文法ですが，詳細については9.1節で説明します．データの型はここではintですが，一般のクラスが使えます．要素数はa(n)のように()で囲んでいます．vectorの初期化法（コンストラクタ）はいくつかあります（次節参照）．aの(j+1)番目の要素は，配列同様a[j]でアクセスできます．

上のコードで注目して欲しいのは，vectorはクラスであり，そのメンバ関数size()がその要素数を戻すことです．よって，a.size()はaの要素数です．vectorの要素数が必要な場合は必ずメンバ関数のsize()を用います．

vectorの要素数：size

vector名.size()

配列と比較してみましょう．配列は，自分の要素数を知らないので，たとえば6.1.1，6.2節のコードのnのように別に要素数を管理する必要がありました．これは，面倒かつ危険です．たとえば，要素数がずれて，データを取りこぼしたり，また，メモリーエラーを起こしたりする間違いは珍しくありません．vectorではsize()を用いれば，こういった危険から解放されます．既に用意されているライブラリ等で，**配列を使う必要がある場合には，vector vの&(v[0])を使うことを勧めます．**配列は初めの要素のアドレスであり（6.7節参照），vectorがこのように配列を含むことは標準で定められています．これにより，vectorの利便性を使いつつ配列を実装する事ができます．

6.8.1 vectorのインデックス

上のコードでは，vectorのインデックスにint型変数を用いましたが，厳密には，下のコードのようにインデックスの型はvectorのsize_typeを用いるべきです．

```
...

for(vector<int>::size_type j=0 ; j<a.size() ; ++j){
  a[j] = j;
}

...
```

コンパイラによっては，vectorのインデックスにint型を用いると警告をします．インデックスが厳密にはint型ではない理由をあげます：

- インデックスは正（0以上）の整数のみ．
- int型変数とsize_typeでは，最大値が異なる（一般には後者の方が大きい）．

vectorのインデックスの型にはsize_typeを用いる事を勧めます．しかし，インデックスにsize_typeを必ず使うのは面倒，というのも理解できます．intを用いても良いですが，上の2点（インデックスは0以上，整数の最大値より大きい配列の場合はsize_type使う）は気に留めておいて下さい．

次のコードはvectorの最大の大きさを確認します．

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

int main(){
  cout<<"int max:\t"<<numeric_limits<int>::max()<<endl;
  vector<int> vInt;
  cout<<"vector<int> max size:\t"<<vInt.max_size()<<endl;
  vector<double> vDouble;
  cout<<"vector<double> max size:\t"<<vDouble.max_size()<<endl;
  vector<string> vString;
  cout<<"vector<string> max size:\t"<<vString.max_size()<<endl;
}
```

プログラムの出力例は以下のとおりです．[1]

```
int max: 2147483647
vector<int> max size: 4611686018427387903
vector<double> max size: 2305843009213693951
vector<string> max size: 576460752303423487
```

limitsのヘッダを読み込むことにより，int型の最大値も得ています．結果を見ると，int，double，string型のvectorで最大要素数は異なり，いずれもint型の最大値より桁違いに大きいことがわかります．実際に使えるvectorの要素数はメモリ容量にも制限され，上の値より一般に小さいです．

6.9 vectorと動的メモリ

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

int main(){
  vector<string> v; //文字列のvector
  string s;
  while(1){
    cout<<"string (xxx to stop) =>  ";
    cin>>s;
    if( s == "xxx" ) {
      break;
    }
    v.push_back(s); //要素を加える
    cout <<"size:\t" << v.size() <<endl;
  }
  for(int j=0 ; j<v.size() ; ++j){
    cout<<v[j]<<" ";
  }
  cout<<endl;
}
```

[1]値は一般にOS，コンパイラに依存します．ここでの結果はgcc 5.4.0 64bit 版で得たものです．

vectorは動的に要素数を変更できます．上のコードはこの1例です．任意の個数の文字列を入力して，データに保持するプログラムです．ここでは，vectorに末尾から1つ要素を加える便利なメンバ関数push_backを使っています．

push_back

```
vector名.push_back(加える要素);
```

vectorの要素数は0からスタートして，入力する度に1増えます．これがわかるように，要素数を入力する度に表示しています．終了の合図が必要なので，xxxを入力したら終了としました．プログラムを走らせた例を示します．

```
string (xxx to stop) =>  This
size: 1
string (xxx to stop) =>  is
size: 2
string (xxx to stop) =>  fun!
size: 3
string (xxx to stop) =>  xxx
This is fun!
```

コンソール入力では事前に要素数がわからないので，この例のようにvectorを使うのは自然です．

6.10 vectorの初期化 — コンストラクタ

vectorはクラスなので初期化にコンストラクタ（4.3節参照）を用います．定義されているコンストラクタでよく使うものを以下にまとめます．

vector初期化1：要素数0

```
vector<データ型> vector名;
```

vector初期化2：要素数指定

```
vector<データ型> vector名(要素数);
```

vector初期化3：要素数と初期値指定

```
vector<データ型> vector名(要素数,初期値);
```

要素が全て同じ初期値になります．

vector初期化4：要素の指定

```
vector<データ型> vector名 = { 要素1, 要素2, 要素3, ...};
```

各要素を指定できます．

データ型にはクラスも使えます．行列（2次元配列）を vectorクラスを使って定義する例を見てみましょう．

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

int main(){
```

```
  const int N(10);
  vector<vector<double> > m(N,vector<double>(N,0.0));
  for(int j=0 ; j<N ; ++j){
    m[j][j] = 1; //対角成分1
  }
  for(int i=0 ; i<N ; ++i){
    for(int j=0 ; j<N ; ++j){
      cout<<m[i][j]<<"\t";
    }
    cout<<endl;
  }
}
```

行列はvectorの要素がvectorの「入れ子」の型で実装できます．この例は，N×N単位行列（対角成分が1，Nは10）を定義し，以下のように表示します．

```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

行列mの実体化の部分を説明します．初期化には上の初期化3の方法を入れ子状に使っています．mをN個の実数型vectorで初期化していて，それぞれが要素数N，値全て0の実数型vectorです．

6.11 関数の値渡しと参照渡し

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

int f(int n){ //値渡しなので変化は関数ブロック内のみ
  n *= -1;
  return n;
}
int main(){
  int p;
  p = 1;
  cout<<p<<", ";
  cout<<f(p)<<", ";
  cout <<p<<endl; //変化していない
}
```

C++の関数の引数は関数内の変数となるので，関数外には影響を及ぼしません（第3章参照）．上のプログラムを走らせると，次の出力を得ます．

```
1, -1, 1
```

ていねいに確認してみましょう．pの値は1，関数内の戻り値は–1で，関数を呼んだ後もpの値は1のままです．関数内では引数に–1をかけて変化させていますが，変化は関数内のみです．関数に引数を与えると，関数が同じ値を持った引数のコピーを作り，それを関数内で使っているからです．このように，関数には値だけしか渡していないので**値渡し**とよびます．

それでは，どうすれば関数を用いて引数を変化させられるのでしょうか？データのアドレスを渡せばできます．これを**参照渡し**とよびます．アドレスの値があれば，引数（アドレス）自体は変更できませんが，指すデータの値を書き換えられるからです（6.3節参照）．C++では，**引数の前に&を付けるだけで参照渡し**になります．上のコード例を参照渡しに書き換えてみましょう．

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

int f(int &n){ //参照渡し．差は&だけ
  n *= -1;
  return n;
}

int main(){
  int p;
  p = 1;
  cout<<p<<", ";
  cout<<f(p)<<", ";
  cout <<p<<endl; //変化している！
}
```

2つのコードの差は関数の引数nの前に&があるだけです．プログラムを走らせると以下の出力が得られます．

```
1, -1, -1
```

先の出力に比べ，最後の値が–1になり，main内で明示的にはpの値を変更していないのに，pの値が変化しています．参照渡しをするには，引数を明示的にデータへのポインタとすることもできます．しかし，C++では&を用いた参照渡しの方が楽で，標準的です．

参照渡し

戻り値データ型 関数名(引数データ型 &変数名,...){...}

参照渡しはいつでも使えます[2]が，どのような状況で参照渡しを使うべきなのでしょうか？使うのが自然な場合の例をあげます．

- 関数，他のオブジェクトに渡すデータが大きい場合：データ量が大きい場合は，コピーをするとメモリを確保するのにも時間がかかり，足りなくなる場合もあります．データが大きく，コピーを作る必要が無い場合は参照渡しが適切です．

- データ初期化などの変更を，関数（他のオブジェクトのメンバ関数を含む）で行う場合：複雑な初期化等は関数を用いた方が整理しやすい場合があります．C++では，データ自体がオブジェクトの一部で，初期化にはそのオブジェクトのメンバ関数を使う場合が多く，その場合には必要ありません．

重要なポイントをあげます．関数や他のオブジェクトにデータを渡し，計算等をする場合に，引数に変更を加えない場合は以下のようにconstを用いるべきです．

[2]実際，Fortran 95のように参照渡ししか無い言語もあります．

<u>引数に変化加えない参照渡し</u>

戻り値データ型 関数名(const 引数データ型 &変数名,...){...}

データに意図しない変更を加えてしまう危険を避けるためです．

参照渡しが適さない場合も多くあります．通常のデータ（実数，整数，大きくないオブジェクト，等々）を引数として関数に渡す場合には，値渡しを使うべきです．関数内で意図せず変更を加える心配が無く，コーディングが楽になるからです．

6.12 vector：応用

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

class Item{
public:
  Item(string s,int p): name_(s),price_(p){}
  int getPrice() const { return price_; }
  void print() const { cout<<name_<<"\t"<<price_<<endl;}
private:
  string name_;
  int price_;
};

int main(){
  vector<Item> list;
  string name;
  int price;
  while(1){
    cout<<"Item, price (price<0 to quit) ? ";
    cin>>name>>price;
    if( price<0 ){
      break;
    }
    list.push_back(Item(name,price));
  }
  int total=0;
  for(vector<Item>::size_type j=0 ; j<list.size() ; ++j){
    total += list[j].getPrice();
  }
  for(vector<Item>::size_type j=0  ; j<list.size() ; ++j){
    list[j].print();
  }
  cout<<"-------------------------------"<<endl;
  cout<<"total:\t"<<total<<endl;
}
```

上のコードは，物品名と価格を入力する「家計簿」アプリケーションで，入力が終わるとリストを出力します．0未満の価格を入力した場合に入力終了するようにしました．入出力例をあげます．

```
Item, price (price<0 to quit) ? 本 3000
Item, price (price<0 to quit) ? 弁当 800
Item, price (price<0 to quit) ? 晩ごはん 1200
Item, price (price<0 to quit) ? a -1
本 3000
弁当 800
晩ごはん 1200
--------------------------------
total: 5000
```

この例では，コンパイル時にはもちろん，入力を始めた時にもアプリケーションには入力されるデータ数はわかりません．このような場合は，vectorが動的にデータ数を増やせるので便利です．vectorのメンバ関数やvectorに使える標準的ツール類を使えば，一部を取り除いたり，後でデータを加えたり，並べ替えたりすることも容易にできます（9.2節参照）．

まとめ：制御

- 配列：同じ型の要素をまとめたデータ
- アドレス（番地）とポインタ（番地の変数）．
  - □ データからアドレス（参照）：&
  - □ ポインタからデータ（デファレンス）：*
- Objectへのpointerはメンバを -> で呼び出す
  ↔ objectは .（ピリオド）で呼び出す．
- newでポインタのデータ実体化，deleteでメモリ解放．
- vector：柔軟性が高い配列のオブジェクト版
  - □ push_back(..)：要素を加える
  - □ size()：要素数
- 配列が必要であればvector vの&(v[0])
- 関数への参照渡し：& を引数名の前に付けるだけ．

コラム 3: 最適化

プログラムの最適化とは，プログラムの実行速度を速くしたり，メモリ消費量を少なくしたりすることです．これは，当然すべきことだと思うでしょう．しかし，偉大なコンピュータ学者のクヌースの以下の素晴らしい言葉を思い出すべきです．[a]

「早すぎる時点での最適化は諸悪の根源である.」

まずプログラミングで一番重要なのは意図通りに正しく作動することです．絶対忘れてはいけません．「最適化」を目指してコードを改変して，正しく作動しなくってしまったりしたら，元も子もありません．まずは，最適化はあまり気にせず，正しく作動するコードを書きましょう．

その昔，コンピュータを使える機会自体も少なかったり，コンピュータが非力で処理速度が遅かった時代には，いかに最適化するかというのは最重要な問題でした．しかし，現在のように，至るところに比較的強力なコンピュータがある時代では，時間については，第1章でも述べたように**自分の時間が一番重要**と考えるべきです．多くの場合，プログラムの処理時間は実質秒単位やせいぜい数時間です．たとえば，数時間かけて，処理時間を２時間から１時間にできたとしても（コードを変更して処理速度が２倍になることはまずありません）全く得をしていません．ましてや，数秒で処理が終わるコードを何時間もかけて速くすることに時間的メリットはありません．

それでは，最適化について考えなくて良いのでしょうか？それも極端です．基本的には，シンプルで無駄の無いコードを書くことは最適化の観点からも，間違いをしにくいという観点からも重要です．常に心がけるべきことです．「シンプルで無駄の無い」の意味は難しいですが，論理がすっきり反映され，無駄な繰り返し等が少なく，無駄なメモリを確保したりしない，ということがあげられます．たとえば，配列，`vector`は動的に確保できるので，無駄ない大きさで作る，といった注意も一例です．一般的に，論理がしっかりして，無駄の無いコードの方がコンパイラも最適化しやすいです．

それでも処理時間が気になる場合は，コンパイラの種類，そしてコンパイラの最適化オプションを色々試しましょう．これは，コードを書き換えなくてもできる簡単な最適化です．最適化のオプション（`-O`，等）を加えるだけで，2倍以上速くなることも珍しくありません．近年のコンパイラは進歩していて，平行化など細かく最適化オプションも指定できます．

最適化に時間をかけるのが有用になってくるのは，処理時間が長いプログラムで（例えば月単位）や，パラメータを変更して何回も同じプログラムを使う場合等です．これは数値計算，シュミレーション等では珍しくありません．このような場合は，コードを合理的に調整することも有意義です．やみくもに「こうすれば速くなるだろう」といった憶測で変更することは勧められません．どこで時間やメモリを使っているのかといったことをしっかり把握する必要があります．そうしないと全く無駄な努力になりかねません．正しく作動しなくなるかも知れませんが，一部を取り除いてみて速さを比べるといったこともできます．これには新たなソフトウェアは必要としません．

本格的に最適化をしようとするのであれば，`gprof`等のプロファイラを活用することを強く勧めます．プロファイラを使うと，どの部分で時間を消費しているのかといったことが特定でき，どの部分のコードを改良すれば良いのか，あるいは場合によっては，これ以上あまり速くならない，といったことがわかります．

[a] “premature optimization is the root of all evil (or at least most of it) in programming.” 「Art of Computer Programming」，Donald Knuth （1968）．

Chapter 7

ファイルと入出力

本章では，ファイルへの出力とファイルからの入力を扱います．

7.1　ファイルへの出力の基本

```
#include <iostream>
#include <fstream> //ファイル入出力に必要
using namespace std;

int main(){
  ofstream fOut("test.dat");//上書き用に開く
  fOut<<"This is a test."<<endl;
  fOut.close();
}
```

上のプログラムを実行するとファイルtest.datに「This is a test.」と書き込みます．出力に<<を用いるのは，コンソールと全く同じです．ファイルに出力するには，ファイル名を指定しなければならないので，それが加わっただけです．

　ファイル出力には，書き込み用にファイルを開く必要があります．ofstream（Output file streamの略)の行で，fOutを**出力ファイルストリーム**（ファイルに出力するオブジェクト）として宣言し，さらに，ファイルtest.datに書き出すオブジェクトとして実体化しています．これにより，指定されたファイルが書き込み用に開かれています．ファイルは，**プログラムを実行したディレクトリ（フォルダともよびます）に作られます．**

```
出力ファイルストリーム（上書き）

ofstream 出力ファイルストリーム名(出力ファイル名);
```

ofstreamはデフォルトではファイルを上書きします．ファイルに追加で書き込むためには，次のようにofstreamのコンストラクタの引数で指定するだけでできます．

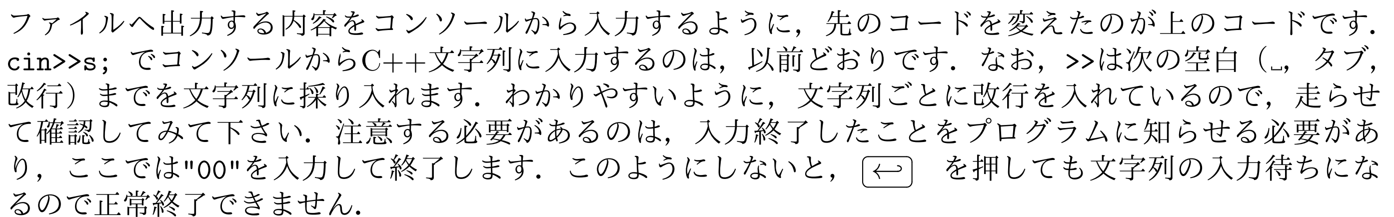
```
出力ファイルストリーム（追加）

ofstream 出力ファイルストリーム名(出力ファイル名,ios::app);
```

コンソールはファイル（のようなもの）とみなせるので，今まで使ってきたcout<<$\cdots$の指示は，コンソールのファイルストリームに書き込んでいて，このファイルストリームは自動的に開かれていた，と理解できます．

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

int main(){
  ofstream fOut("test.dat");
  string s;
  while(1){
    cout<<"text  (\"00\" to stop) => ";
    cin>>s;
    if( s == "00" )
      break;
    fOut<<s<<endl;
  }
  fOut.close();
}
```



ファイルへ出力する内容をコンソールから入力するように，先のコードを変えたのが上のコードです．`cin>>s;` でコンソールからC++文字列に入力するのは，以前どおりです．なお，`>>`は次の空白（␣，タブ，改行）までを文字列に採り入れます．わかりやすいように，文字列ごとに改行を入れているので，走らせて確認してみて下さい．注意する必要があるのは，入力終了したことをプログラムに知らせる必要があり，ここでは`"00"`を入力して終了します．このようにしないと，⏎ を押しても文字列の入力待ちになるので正常終了できません．

7.2 ファイルからの入力

7.2.1 ファイルからの入力の基本

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

int main(){
  ifstream fIn("test.dat");
  string s;
  while(1){
    fIn>>s;
    if( s == "00" )
      break;
    cout<<s<<endl;
  }
}
```

ファイルからの入力ををコンソールに出力するコードです．入力部分は，前節のコンソールからの入力のコードの`cin`を`ifstream`の`fIn`に変更しただけでです！わかりやすいでしょう．`ifstream`は入力ファイルストリーム（Input file streamの略）です．`ofstream`と同様に，`ifstream fIn("test.dat");`の指示で`test.dat`から読み込む入力ファイルストリーム`fIn`を開きます．このコードでは`"00"`で入力終了を検出しています．よって，ファイルにこの文字列が無いと，プログラムは終了しません．このような書き方は危険なので，ファイルの終わりの検出にはこのような方法は**通常用いません**．これについては次のコードで説明します．一方，読み出すデータの**区切り**に目印の文字列を使うのは珍しくありません．

7.2.2 >>を用いた読み出し

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

int main(){
  ifstream fIn("test.dat");
  string s;
  while( fIn>>s ){
    cout<<s<<endl;
  }
}
```

上のコードのように，ファイルの終わりを自動検出する方法は実践的です．>>演算はファイル終わりに到達すると，偽の値を戻すので，ファイル終わりに到達するとwhileループを出ます．以前指摘したように，>>は次の空白の前まで読むので，ファイルの中身が

```
This is a test.
```

であった場合，プログラム出力は以下の通りです．

```
This
is
a
test.
```

7.2.3 getline を用いた読み出し

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

int main(){
  ifstream fIn("test.dat");
  string s;
  while( getline(fIn,s) ){
    cout<<s<<endl;
  }
}
```

>>を代わりにgetlineを用いてファイルを読み込むコードです．このプログラムの出力は次のとおりです．

```
This is a test.
```

>>とは異なり，getlineは関数名どおりファイルから1行読み込むことがわかります．getlineは>>と同様にファイルの終わりで偽を戻し，検出できるので，これをwhile等の条件に使えます．

ファイル入出力を扱ってきました．一般に出力は書き手の意図に従って「書きっぱなし」なので単純です．ファイルからの入力は，ファイル形式を把握して，データを読み出す事が必要です．

7.3 ファイル入出力のポイント

7.3.1 ファイル名の指定

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

int main(){
  cout<<"Filename? => ";
  string s;
  cin>>s;
  ofstream fOut(s);
  fOut<<"This is a test."<<endl;
}
```

上では7.1節のコードをファイル名を入力するように変更しましたが，それ以外は同じです．ファイル名のstringをofstream， ifstreamのコンストラクタ引数に使えば良いだけです．[1]

7.3.2 ファイルを開くことができているかの確認

```
#include <iostream>
#include <fstream> //ファイル入出力に必要
using namespace std;

int main(){
  const string filename("test.dat");
  ofstream fOut(filename); //上書き用に開く
  if(fOut.is_open()){//開けられているか確認
    fOut<<"This is a test."<<endl;
    fOut.close();
  }
  else{
    cerr<<"Can not open file: "<<filename<<endl;
  }
}
```

上のコードでは，ファイルに出力する際に実際にファイルが開けているか確認してから書き込んでいます．このように確認しないと，ファイルに書き込めず，明示的にエラーも出ないので，重要なデータを書き込み損ねる場合もあります．ファイルから入力する際も同様で，データを読み込めていないのに，エラーメッセージも無くて気づかない，といった事態が生じ得ます．

実践的なコードでは，実際に出力，入力，どちらの場合もファイルを開くことができるのか確認することが重要な場合が多々あります．ファイルを入出力のために開けようとして，できない典型的な場合の例をあげます．

- 読み込もうとしているファイルが無い，あるいは読む権限が無い．
- 書き込む権限が無いディレクトリ（存在しないディレクトリを含む）にファイルを作ろうとしている．
- 上書き権限が無いファイルを上書きしようとしている．

ファイルが開くことができているかの確認は簡単なので，重要な場合は必ず確認をしましょう．

[1] C++11標準の前は，ofstream， ifstreamのコンストラクタ引数にはC++ stringは使えず，C文字列である必要がありました．この場合はstringからC文字列を戻すメンバ関数，c_str() を用います．上の例ではsのかわりにs.c_str()を使うことになりますが，これでもコードは（面倒ですが）正しいです．C文字列は文字（char）の配列で，要素は文字列の各文字と'\0'になります．たとえば，“abc”はC文字列として'a','b','c','\0'の4個の文字の要素を持つ文字配列になります．

7.3.3 close()の重要性

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

int main(){
  const string fname("test.dat");
  ofstream fOut(fname);
  fOut<<"Hello World!";
  fOut.close(); //これが無いと文字列表示されない
  ifstream fIn(fname);
  string s;
  while(getline(fIn,s)){
    cout<<s<<endl;
  }
}
```

上のプログラムを走らせると，以下の出力を得ます．

```
Hello World!
```

上の文字列をファイルtest.datに書き出し，それを読み出して標準出力に表示するコードなので，当然に感じるでしょう．ただ，このコードで，fOut.close()は重要です．これが無いと，プログラムを走らせても出力は一般にありません．test.datへの書き込みは，ファイルを閉じる（close）するまでになされますが，閉じていないと，書き込まれている保証はないからです．プログラム内で，ファイルを書き込み，それを読み出す場合には，この点に注意が必要です．ファイルを書き込んで，それでプログラムが終わる場合には，プログラムを終えるときに，ファイルストリームがスコープから外れ，デストラクタが稼働し，ファイルが自動的に閉じられるのでこの重要性に気づかない場合が多いでしょう．気づいたかも知れませんが，出力のファイルストリーム（ofstream）の場合には明示的にcloseすることが必要な場合がありますが，ifstreamの場合には，通常は明示的に閉じる必要はありません．

7.4 ファイル入出力：応用

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

class Data{
public:
  Data(string s){ readFile(s); }
  void readFile(string);
  vector<vector<double> > getData() const { return data_; }
  vector<string> getComment() const { return comment_; }
  bool isRectangular() const; //行ごとデータ数一定？
private:
  vector<vector<double> > data_;
  vector<string> comment_;
};
```

```
void Data::readFile(string fileName){
  ifstream fIn(fileName);
  if( !fIn.is_open() ){
    cerr<<"Can NOT open "<<fileName<<endl;
    terminate();
  }
  string line;
  while(getline(fIn,line)){
    if( line[0] == '#' ){
      comment_.push_back(line);
    }
    else{
      stringstream ss(line);
      vector<double> v;
      double x;
      while(ss>>x){
v.push_back(x);
      }
      data_.push_back(v);
    }
  }
}

bool Data::isRectangular() const{
  if( data_.size() == 0 ){
    cerr<<"no data!"<<endl;
    terminate();
  }
  for(vector<vector<double> >::size_type j=1;
     j<data_.size() ; ++j){
    if( data_[j].size() != data_[0].size() ){
      return false;
    }
  }
  return true;
}

int main(){
  cout<<"Data file name ? =>  ";
  string s;
  cin >> s;
  Data dataObj(s);
  vector<vector<double> > data(dataObj.getData());
  for(  vector<vector<double> >::size_type j=0 ;
          j<data.size() ; ++j){
    for( vector<double>::size_type k=0 ;
           k<data[j].size() ; ++k){
      cout<<data[j][k]<<"\t";
    }
    cout<<endl;
  }
```

```
   cout<<"Data is "<< (dataObj.isRectangular() ? "" : "NOT " ) << "rectangular"<<endl;
   cout<<"Comments:"<<endl;
   vector<string> comment(dataObj.getComment());
   for(vector<string>::size_type j=0 ;
         j<comment.size() ; ++j){
     cout<<comment[j]<<endl;
   }
 }
```

上は2次元のデータをファイルから取り込むコード例です．ファイル名を引数としたコンストラクタだけを装備していますが，他のコンストラクタは必要に応じて加えれば良いでしょう．データファイルtest.datの内容が以下の例を考えます．

```
# This is a comment
1.1 2.2 3
4 5
# This is another comment
6 7 8
```

プログラムの出力は以下のとおりです．

```
Data file name ? =>  test.dat
1.1     2.2     3
4       5
6       7       8
Data is NOT rectangular
Comments:
# This is a comment
# This is another comment
```

メンバ関数readFileは，まずデータファイルが読み込めるかチェックし，ファイルからデータを1行ずつ読み込みます．ここでは，コメントを#で始まる行として想定しています．コメント行でない場合は，数をvector<double>にpush_backで取り込みます．ここではそこまでしていませんが，数字として読み込めるかチェックすれば，より安全なコードになります．コメントは別のvector<string>に追加していきます．各行のデータ数が同じであるかどうかをチェックするメンバ関数isRectangularも装備しました．コードでは，読み込んだデータとコメントをコンソールに出力します．行毎のデータ数が同じであるかもチェックしています（上の例では同じではありません）．

このコードでは，何か問題があれば，terminateで終了する仕様となっています．少し乱暴に感じるかも知れませんが，間違ったことをするよりは，何もしない方が良いという方針です．方針は状況によって変更すべきです．

(条件 ? x : y) 上のコードでは今まで扱っていなかった文法を次の部分で用いています．

```
(dataObj.isRectangular() ? "" : "NOT " )
```

これは，一般に次の形式をとり，条件が真であればx，そうでなければyとなります．

```
( 条件 ? x : y )
```

同じ内容をif，elseを用いて書くこともできますが，この簡略形の方が楽です．慣れれば直観的にも読み取りやすいので，よく使われる文法です．

まとめ：ファイルと入出力

- `#include <fstream>`：ファイル入出力に必要なヘッダ.
- `ofstream`：出力ファイルストリーム，デフォルトは上書き（追加は`ios::app`）.
- `<<`：出力
- `ifstream`：読み込むファイル.
- `>>`：次の空白まで読み込む，`getline`：1行読み込む.

その他

- `( 条件 ? x : y )`：条件が真ならx，偽ならy

Chapter 8

継承

本章では，継承について学びます．継承を用いれば，1つのクラスに機能を付け加えた新たなクラスを定義できます．これは自然な発想であるとともに便利な機能ですが，使い方には注意が必要です．自分から継承を積極的に使う意図が無くてもGUIアプリケーション開発など，既存のフレームワークを活用する際には継承を使う必要があります．

8.1 継承の目的と概念

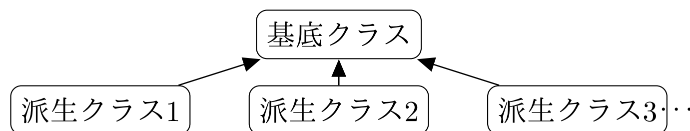


Figure 8.1: 継承：基底クラスから派生したクラスには基底クラスの機能に加え，新たな機能をもたせられる．

コードを分析する前に，なぜ継承を使うのか，あるいは使わないのか，について考えてみましょう．クラスを使ってコーディングしていると，既存のクラスに機能を加えた新たなクラスを使いたくなる場面は多くあります．その場合，大きく分けて3つの解決策があります．

I 既存クラスをコピーし，改変したクラスを作る

II 既存クラスを継承したクラスを作る

III 既存クラスをメンバにしたクラスを作る

それぞれのアプローチを比較してみましょう：

I：コピーして改変するのは，原始的で，考え方としては簡単です．そして，それが実用的な場面も多くあります．他の独立したプロジェクトで用いたクラスを改変して使う場合等は，自然な解決策です．しかし，同じプロジェクト内で同じクラスを拡張したクラスを多くコピーして定義することは管理の手間を含め，無駄が多くなりがちです．また，異なるプロジェクト間でもヘッダファイル（9.7節参照）を用いて，クラスを共有することも可能で，その方が適切な場合もあります．

II：継承（図8.1参照）は，既存のクラス（**基底クラス**）にデータ，機能を付け加え，新たなクラス（**派生クラス**）を作る方式です．派生クラスは基底クラスを**継承**している，といいます．文法は以下で詳しく説明します．理屈は自然でわかりやすいでしょう．付け加えた部分だけを新たにコーディングするだけなので，無駄もありません．ただし，基底クラスの変更は全ての派生クラスに影響を及ぼします．よって，基底クラスには，派生クラスのデザインや使い方を考慮したデザインが必要で，注意を要します．

III：クラスを拡張せずに，そのクラスのオブジェクトを新たなクラスのメンバに含めれば，そのデータ，機能をメンバのデータ，機能として取り込めます．これは，プログラミング手法としては，単純で，多く使われます．

コピーをしないのであれば，**II**と**III**からの選択になります．**II**を“Is a”，**III**を“Have a”の関係とよぶこともあります．継承の概念を聞くと，自然に感じて，多くの場面で継承を使いたくなるかも知れませ

ん．しかし，必ずしも継承ばかりが自然とは限りません．たとえば，動物のクラスを考えてみましょう．キリン，ライオンは動物の1種（“Is a”）なので，動物クラスを継承したクラスを作るしかない！と考えるかも知れません．しかし，キリン，ライオンは動物の性質を持っているので（“Have a”），動物の性質のクラスをメンバとして持つクラスとしても考えられます．どちらの方が良いのでしょうか．一般には，**III**のクラスをメンバに加える方法の方が単純でデザインも管理も楽なので，それで十分であれば，**III**のアプローチを勧めます．**II**の継承が必要となる場合には，**ポリモーフィズム**（多態性，8.6節参照）が求められる場合や，活用したいフレームワークの仕様によって必要になる場合などがあります．この問題について，本章の最後で継承について学んだ後にまた考えます．

ここで，共通のコードを用いる方法に対して，コピーして繰り返して使う方法について比較してみましょう．コードを書き写して，改変をしていくことは必ずしも「悪い」わけではありません．ただ，一般にコードの繰り返しを避け，共通のコードを利用できる場合は使うようにします．共通のコードを利用するには，上のようなクラスの拡張法**II,III**ばかりではなく，今まで扱ってきた関数やクラスを用いて，共通の処理を行う場合も含まれます．共通のコードを用いる，つまり抽象化することには，以下の様な利点があります．

- コードに無駄が少なく，わかりやすい．よって間違いもしにくい．
- コードの改良や訂正を一回すれば，全てに反映される．
- 不要な繰返しが無い方がコンピュータにとっても効率的（コンパイラも最適化しやすい）．

コーディング全般について言えることですが，繰り返しが多い場合は，デザインを見つめ直して，よりシンプルなコードを目指すことを勧めます．

8.2 継承の基本

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

class B{ //基底クラス
public:
  void fB(){ cout<<"B function"<<endl; }
};

class D: public B{ //派生クラス
public:
  void fD() const { cout<<"D function"<<endl; }
};

int main(){
  D dObj; //派生クラスオブジェクト
  dObj.fD(); //派生クラスのメンバ関数
  dObj.fB();//基底クラスのメンバ関数
}
```

上のプログラムの実行結果は以下のとおりです．

```
D function
B function
```

コードでは基底クラス`B`，派生クラス`D`を定義し，それぞれにメンバ関数`fB`，`fD`を定義しています．派生クラスのオブジェクト`dObj`を実体化します．`D`のオブジェクトが派生クラス`D`のメンバ関数を呼べるのは当然です．派生クラスでは明示的に定義していない基底クラス`B`のメンバ関数も呼べることから，基底クラスの機能を継承していることがわかります．基底クラスを基本クラス， スーパークラスや親クラスと呼ぶ場合もあります．

継承を使うための文法は，以下のとおりです．

継承

```
class 派生クラス名 : public 基底クラス名 { ...（クラスを定義）...
};
```

ここで説明している継承はpublic継承とよびます．public継承のアクセスコントロールは以下のとおりです．

1. 基底クラスのpublicメンバは派生クラスのpublicメンバ．
2. 基底クラスのprotectedメンバは派生クラスのprotectedメンバ．
3. 基底クラスのprivateメンバは派生クラスからアクセスできない．

直観的にわかりやすいと思います．privateメンバがそのクラス自身からしかアクセスできないのは，既に説明したとおりで，継承に限ったことではありません．以下ではpublic継承しか使いませんが，それ以外に，protected, private継承があり，使うためには，上の宣言の`public`の部分をそれぞれ`protected`，`private`に変更します．protected継承では，基底クラスのpublic, protected メンバは派生クラスのprotectedメンバになり，private継承では，基底クラスのpublic, protected メンバは派生クラスのprivateメンバになります．

また，ここでは扱いませんが，複数の基底クラスから継承する派生クラスを定義することもできます．これを**多重継承**とよびます．コードが複雑になりやすいので，必要な場面のみで使うべきです．

継承の重要なポイントをあげます．

- 派生クラスは基底クラスのpublicとprotectedであるメンバ関数，メンバデータを備えている．
- 派生クラスは基底クラスのメンバに加えてメンバ関数，メンバデータを定義できる．
- 派生クラスは基底クラスのメンバをオーバーライド（上書き）できる．

最後の点についてはこれから説明します．

8.3 継承とvirtual関数

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

class B{ //基底クラス
public:
  virtual void fB() const { cout<<"B function"<<endl; }
  void setB(string s){ sB = s; }
  string getB() const { return sB; }
  virtual ~B(){} //基底クラスには必ずvirtualデストラクタ
private:
  string sB;
};
class D: public B{ //派生クラス
public:
  void fD() const { cout<<"D function"<<endl; }
  void fB() const { cout<<"Override fB from D"<<endl;} //オーバーライド
};

int main(){
  D dObj;
  dObj.setB("string in B"); //基底クラスのメンバ関数を派生クラスで使う
```

```
  cout<<dObj.getB()<<endl;  //
  dObj.fB();  //オーバライド確認
}
```

継承の概念を確認しつつ，**virtual関数**（仮想関数ともよびます）を導入したのが上のコードです．プログラムを走らせると以下の出力が得られます．

```
string in B
Override fB from D
```

前節の例と同様に，クラスDはクラスBを継承しています．変数sBは基底クラスのprivate メンバデータです．よって，B内からしか直接はアクセスできません．派生クラスでは基底クラスのpublicメンバ関数setB，getBが使え，それらを通じて，sBを操作できます．

1つ新しいのは，基底クラスのpublic関数fBを，引数，戻り値の型が同じ派生クラスのメンバ関数fBが上書きしていることです．上書きしていることは，出力から確認できます．この上書きを**オーバーライド**とよびます[1]．オーバーライドする関数は引数の数，型，戻り値の型，さらにconstなどのvirtual以外の修飾子，の全てが元の関数と同じでなければなりません．

オーバーライドされる基底関数はvirtual関数にする

ことに気を付けなければなりません．理由は（8.6節参照）で詳しく説明します．クラスにはデストラクタが必ず装備されていますが（明示的に定義しなくてもコンパイラが装備します），基底クラスのデストラクタは必ずvirutal関数にすべきです．必ずオーバーライドされるからです．上に~Bのコードの例あげました．定義は必要ですが，中身は書く必要はありません．デストラクタについて考えると，コンストラクタも気になるかも知れませんが，コンストラクタはvirtual関数にできません．

override宣言　派生クラスでオーバーライドした関数の関数名と修飾子の後にoverrideのキーワードを加えることで，virtual関数をオーバーライドしているとを明示的に宣言することができます．たとえば，上のコードでは，派生クラスの関数fBを以下のように宣言します．

```
  void fB() const override { cout<<"Override fB from D"<<endl;}
```

これ以外は全く同じです．既に指摘したように，修飾子を含め同じ関数でないとオーバーライドできないので，オーバーライドしているつもりになっていてしていない場合も起こりやすいです．override宣言によって，virtual関数をオーバーライドできていない場合にはコンパイル時のエラーになり，間違いを早く見つけやすくなるので使う事を勧めます．

override宣言（インライン）

派生クラスメンバ関数名(引数) 修飾子 override { 指示; }

override宣言（クラス宣言外で関数定義）

派生クラスメンバ関数名(引数) 修飾子 override;

クラス宣言外でメンバ関数を定義する場合は，override宣言はクラス宣言内で使い，メンバ関数の定義には付けません（8.8節の例参照）．

8.4 継承とコンストラクタ

[1]オーバーライドとオーバーロード（3.4節参照）を混同しないようにしましょう．オーバーライドは関数の上書きで，オーバーロードは引数が異なる，同名の別の関数を定義することです．

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

class B{ //基底クラス
public:
  B(){ cout<<"B constructed"<<endl; }
  virtual ~B(){cout <<"B destructed"<<endl; }
};
class D: public B{ //派生クラス
public:
  D(){ cout<<"D constructed"<<endl; }
  ~D() override {cout <<"D destructed"<<endl; }
};

int main(){
  D d;
}
```

上のプログラムを走らせると，以下の出力を得ます．

```
B constructed
D constructed
D destructed
B destructed
```

上のコードはコンストラクタ，デストラクタしか無く，どのタイミングでそれらが呼び出されるかを確認しています．派生クラスを実体化すると，まず，基底クラスが実体化され，それから派生クラスが実体化されます．元の部分が作られ，拡張部分が追加されるので納得できるでしょう．逆に，破棄される際は，逆の順序になり，これも自然です．コンストラクタとデストラクタはクラスの基本であり，今までどおりに使えます．

引数がある基底クラスコンストラクタ　上で見たように派生クラスを実体化すると，派生クラスのコンストラクタが呼び出され，さらにそこから基底クラスのコンストラクタが呼び出されます．上の例のように基底クラスのデフォルトコンストラクタ（引数が無いコンストラクタ）を使う場合は，自動的なので良いですが，引数がある基底クラスのコンストラクタを使う場合には注意が必要なので説明します．

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

class B{ //基底クラス
public:
  B(string s,int n): bs_(s),bn_(n){}
  virtual void print() const { cout<<bs_<<", "<<bn_; }
  virtual ~B(){}
private:
  string bs_;
  int bn_;
};
class D: public B{ //派生クラス
public:
  D(string s,int a,double x) : B(s,a),dx_(x){}
  void print() const override { B::print(); cout<<", "<<dx_;}
private:
```

```
  double dx_;
};

int main(){
  D d("hello",1,2.2);
  d.print();
  cout<<endl;
}
```

上のプログラムを実行した結果は次です．

```
hello, 1, 2.2
```

上のコード例では，B,Dのクラスはともに実体化する際に引数でメンバ変数を初期化しています．オブジェクトのメンバ変数を初期化しない状態で実体化しないようにする場合は多く，そういった形式の例です．基底クラスのコンストラクタは今までどおりです．基底クラスのメンバデータがprivateである場合には派生クラスは直接に基底クラスのデータは初期化できません．上のコードのように，基底クラス名を用いてデータ初期化の一部で呼び出します．直観的にはわかりやすいですが，基底クラスのコンストラクタを派生クラスのコンストラクタとして呼び出すためには上のような文法を使う必要があります．たとえば，上のコードで，派生クラスのコンストラクタのブロック内（{}内）でコンストラクタBを呼び出すと文法エラーになります．派生クラスのコンストラクタブロックは派生クラスが実体化されてから実行され，そのために基底クラスの実体化が必要だからです．

派生クラスコンストラクタから引数のある基底クラスコンストラクタの呼び出し方

派生クラス名(引数) : 基底クラス名(引数) { 派生クラスコンストラクタ内指示 }

上の状況では，使いたい基底クラスのメンバデータをprotectedにして，派生クラスから直接アクセスすることによっても同じ目的を達成できます．どちらの方が良いかは状況によりますが，上の例のように，基底クラスのコンストラクタを使った方が適切な場合は多いです．

派生クラスからオーバーライドされた基底クラスのメンバ関数を呼び出す 上のコードでは，もう1つ重要な文法のポイントがあります．派生クラスのメンバ関数からオーバーライドされた基底クラスのメンバ関数をB::print(); で呼び出しています．オーバーライドしているメンバ関数はデフォルトでは派生クラスの同名のメンバ関数を呼び出すので基底クラスの関数であることをこのように明示する必要があります．

派生クラスからオーバーライドされた基底クラスのメンバ関数の呼び出し方

基底クラス名::関数名(引数)

8.5 継承とポインタ

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

class B{ //基底クラス
public:
  virtual void fB() const { cout<<"B function"<<endl; }
  void setB(string s){ sB = s; }
  string getB() const { return sB; }
  virtual ~B(){} //基底クラスには必ずvirtualデストラクタ
private:
  string sB;
```

```
};

class D: public B{ //派生クラス
public:
  void fD() const { cout<<"D function"<<endl; }
  void fB() const override { cout<<"Override fB from D"<<endl;} //オーバーライド
};

int main(){
  D *dp;//派生クラスポインタ
  dp = new D;
  dp->setB("string in B"); //基底クラスのメンバ関数を派生クラスで使う
  cout<<dp->getB()<<endl;  //
  dp->fB();  //オーバライド確認
  delete dp;
}
```

前節のコードを派生クラスオブジェクトへのポインタを用いて書き換えたのが上のコードです．プログラムの作動は前節のコードと同じです．オブジェクトのポインタについては既に学んだので，新しい要素はありませんが，次節でポインタの新しい使い方を説明する前に確認しておきます．

8.6 ポリモーフィズム（多態性）

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

class B{ //基底クラス
public:
  virtual void f(){ cout<<"Base"<<endl; }
  virtual ~B(){} //基底クラスには必ずvirtualデストラクタ
private:
  string sB;
};
class D1: public B{ //派生クラス
public:
  void f() override { cout<<"Derived 1"<<endl; }
};
class D2: public B{ //派生クラス
public:
  void f() override { cout<<"Derived 2"<<endl; }
};

int main(){
  B *bp; //基底クラスポインタ
  bp = new B;
  bp->f();
  delete bp;
  bp = new D1;//基底クラスポインタは派生クラス指せる
  bp->f();
  delete bp;
  bp = new D2;
```

```
  bp->f();
  delete bp;
}
```

上のコードでは，基底クラスBとそれを継承したクラスを2つ，D1，D2を定義し，Bクラスのポインタbpを定義しています．プログラム出力は以下です．

```
Base
Derived 1
Derived 2
```

基底クラスのポインタbpが基底クラスのオブジェクトを指して，メンバ関数を実行しています．そして，**基底クラスのポインタは派生クラスオブジェクトを指せる**ので，D1，D2のオブジェクトを同じポインタbpで順番に指しています．上の例では，1つのポインタでB，D1，D2の3種のオブジェクトのふるまいができています．これが，**ポリモーフィズム**です．多態性，あるいは多相性とよぶこともあります．今までは，ポインタを使うかにかかわらず，どのようなデータもその型を指定し，その型のメンバしか使えませんでした．継承と基底クラスのポインタを使うことにより，1つのポインタが，複数のクラスのオブジェクトを指せます．しかも，どのクラスを指すかは**実行時に指定**できます．基底クラスのポインタがどの派生クラス，あるいは基底クラスのオブジェクトを指すかは，実行時の入力内容や乱数で決めることもできます．

ところで，ポリモーフィズムはどのような場面で便利なのでしょうか？たとえば，複数の派生クラスがある場合，それをまとめたvector（あるいは配列）をポリモーフィズムを使って，基底クラスのポインタの配列として作れます（8.8節参照）．ポリモーフィズムを使わなければ，複数のvectorを管理する必要があり，一般には順序が入り乱れているので，管理は面倒です．また，ゲームで怪獣それぞれをクラスを使って定義した場合を考えてみましょう．出てくる怪獣がランダムであれば，ポリモーフィズムを使った方が自然にコーディングできる場合もでてきます．ポリモーフィズムに関連して，重要な点を述べます．

基底クラスのポインタは派生クラスのオブジェクトを指せる． 上のコードのクラスの具体例で，ポインタがオブジェクトを指せる場合と指せない場合を○×で表示すると次の表になります．

	Bオブジェクト	D1オブジェクト	D2オブジェクト
Bポインタ	○	○	○
D1ポインタ	×	○	×
D2ポインタ	×	×	○

クラスのポインタがそのクラスを実体化したオブジェクトを指せるのは当然で，それ以外は，基底クラスのポインタが派生クラスのオブジェクトを指すことができるだけです．これがポリモーフィズムの基本的な仕組みです．考えてみると，派生クラスは，基底クラスを拡張したものなので，基底クラスのポインタの持つ機能全てに対応できるので，派生クラスオブジェクトを指せるのは自然です．基底クラスのポインタは基底クラスで定義されたメンバデータ，メンバ関数（オーバーライドされたものを含む）にアクセスできます．

基底クラスのポインタは派生クラス独自のメンバデータ，メンバ関数にはアクセスできない． 基底クラスポインタは基底クラスのメンバしか知らないので，派生クラスで付け加えたメンバにはアクセスできません．これは当然でしょう．

派生クラスのポインタは基底クラスのオブジェクトを指せない． 上の表にもこの点は示されています．派生クラスのポインタは，派生クラスの機能を持っていますが，これは基底クラスでは定義されていないものも一般に含みます．よって，それを装備していない基底クラスのオブジェクトを指すと矛盾が生じます．

基底クラスで上書きされる可能性のあるメンバ関数はvirtual関数にする．

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

class B{ //基底クラス
public:
  virtual void fBVirtual() const { cout<<"B function (virtual)"<<endl; }
  void fBNonVirtual() const { cout<<"B function (NOT virtual)"<<endl; }
  virtual ~B(){} //基底クラスには必ずvirtualデストラクタ
private:
  string sB;
};
class D: public B{ //派生クラス
public:
  void fBVirtual()  const override { cout<<"D function (virtual)"<<endl; }
  void fBNonVirtual() const { cout<<"D function (NOT virtual)"<<endl; }
};

int main(){
  D dObj;//派生クラスオブジェクト
  D *dp; //派生クラスポインタ
  B *bp; //基底クラスポインタ
  dp = new D;
  bp = new D;
  dObj.fBVirtual();  //オーバライド確認
  dp->fBVirtual();
  bp->fBVirtual();
  dObj.fBNonVirtual();
  dp->fBNonVirtual();
  bp->fBNonVirtual();//基底クラスのメンバ関数！
  delete dp;
  delete bp;
}
```

上の具体例を通じてなぜオーバーライドされる関数はvirtualであるべきか考えてみましょう．このプログラムの出力結果は次のとおりです．

```
D function (virtual)
D function (virtual)
D function (virtual)
D function (NOT virtual)
D function (NOT virtual)
B function (NOT virtual)
```

派生クラスDで基底クラスBのvirtualとそうでないメンバ関数，fBVirtual，fBNonVirtual，をオーバーライドしています．この両方を，Dのオブジェクトd0bj，派生クラスのポインタdp，基底クラスのポインタbpでよびます．出力の最後の行からわかるように，基底クラスのポインタでよぼうとした場合だけは，オーバーライドした関数ではなく，基底クラスのメンバ関数を呼び出してしまいます．つまり，virtual関数にしないと派生クラスのオブジェクトは派生クラスのポインタ，派生クラスのオブジェクトとしてアクセスする場合と基底クラスのポインタでアクセスする場合でふるまいが異なります．これは非常に紛らわしいです．さらに，virtual関数を用いない限り基底クラスのメンバ関数をよびだすので，ポリモーフィズムは実現できません．以上がオーバーライドされる可能性のあるメンバ関数をvirtual関数にすべき理由です．

なぜ上のようなふるまいをするのか考えてみると，本来，基底クラスのポインタは基底クラスのメンバ関数を呼び出します．むしろ，それをしないために指示（virtual）が必要なわけです．

ポリモーフィズムの条件

- 基底クラスのポインタで派生クラスを指す
- 基底クラスのメンバ関数はvirtual関数

基底クラスのデストラクタはvirtual関数にする．

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

class B{ //基底クラス
public:
  virtual ~B(){cout<<"B destructor"<<endl;}
};
class D: public B{ //派生クラス
public:
  virtual ~D(){cout<<"D destructor"<<endl;}
};

int main(){
  B *bp;
  bp = new D;
  delete bp;
}
```

上のコード例でデストラクタの挙動を確かめてみましょう．

```
D destructor
B destructor
```

基底クラスのポインタで派生クラスのオブジェクトを指しています．`delete`でオブジェクトを破棄すると，`D`，`B`のデストラクタがこの順序で呼ばれます．これが正しい挙動です．まず，派生クラスの付け加えた分（メモリ等）の後始末をして破棄し，その後で，基底クラスのオブジェクトを破棄しています．

基底クラスのポインタで派生クラスのオブジェクトを指した場合，もしも**デストラクタがvirtual関数でないと，基底クラスのデストラクタだけが呼ばれます．**これは，すぐ前に説明した`virtual`関数であるかどうかの差です．上のコードで `B`の前の`virtual`を取り除けば，`B destructor`のみ出力され，基底クラスのデストラクタしか呼ばれないことが確認できます．そうすると，基底クラス分のオブジェクトが破棄されるだけで，派生クラスオブジェクトの後始末はされません．よって，不要なメモリの一部が解放されない，等の問題が生じ得ます．これが基底クラスのデストラクタをvirtual関数にすべき理由です．

8.7 抽象クラス

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

class B{ //基底クラス
public:
  virtual void fB() const = 0;
  void setB(string s){ sB = s; }
  string getB() const { return sB; }
```

```
  virtual ~B(){} //基底クラスには必ずvirtualデストラクタ
private:
  string sB;
};
class D: public B{ //派生クラス
public:
  void fD() const { cout<<"D function"<<endl; }
  void fB() const override { cout<<"Override fB from D"<<endl;} //オーバーライド
};

int main(){
  D dObj;
  dObj.setB("string in B"); //基底クラスのメンバ関数を派生クラスで使う
  cout<<dObj.getB()<<endl;  //
  dObj.fB();  //オーバライド確認
}
```

上のコードは8.3節のコードとの差は，基底クラスのメンバ関数の定義が`virtual void fB() const = 0;`であるだけです．全く同じ作動をします．このように`= 0;`と定義された関数は，**pure virtual関数**とよび，このメンバ関数は**よべません．**よって，**基底クラスBは実体化できません．**これが，pure virtual関数を持つか持たないかの差です．8.3節のコードではBも実体化できました．このように，pure virtual関数をメンバ関数に持つクラスを**抽象クラス**とよび，抽象クラスは実体化できません．

pure virtual関数の定義

戻り値型 関数名（引数型）`= 0;`

抽象クラスは，それを継承した派生クラスを作って初めて意味があります．抽象クラスを継承したクラスでは，pure virtual関数は必ずオーバーライドする必要があります．たとえば，上のコードでは，クラスDのメンバ関数fBの定義が無ければコンパイルエラーとなります．

動物のクラスを継承した犬，猫等のクラスがある場合を考えましょう．犬，猫の実体は存在しても，動物の実体は存在しなくて良いでしょう．そう考えると抽象クラスは，整理の仕方として自然な発想でしょう．

8.8 継承：応用

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

class Item{
public:
  virtual void input();
  int getPrice() const { return price_; }
  virtual void print() const { cout<<name_<<"\t"<<price_<<endl;}
  virtual ~Item(){}
protected:
  string name_;
  double price_;
};
void  Item::input(){
    cout<<"Item, price  ? ";
```

```
    cin>>name_>>price_;
}
class ItemWeight : public Item{
public:
  void input() override;
  void print() const override { cout<<name_<<"\t"<<price_<<"\t"<<weight_<<endl;}
private:
  double weight_;
};
void  ItemWeight::input(){
  cout<<"Item, price, weight [kg] ? => ";
  cin>>name_>>price_>>weight_;
}

int main(){
  vector<Item *> list;
  while(1){
    string choice;
    cout<<"0: exit, 1: item with weight, other: Item => ";
    cin>>choice;
    Item *pItem;
    if( choice == "0" ){ //入力時に型を判断
      break;
    } else if (choice == "1" ){
      pItem = new ItemWeight;
    } else{
      pItem = new Item;
    }
    list.push_back(pItem);
    pItem->input();
  }
  int total=0;
  for(vector<int>::size_type j=0 ; j<list.size() ; ++j){
    total += list[j]->getPrice();
  }
  cout<<"item\tprice\tweight[kg]"<<endl;
  for(vector<int>::size_type j=0 ; j<list.size() ; ++j){
    list[j]->print();
  }
  cout<<"------------------------------"<<endl;
  cout<<"total:\t"<<total<<endl;
}
```

少し長いですが，6.12節のコードを拡張し，継承を使ったポリモーフィズムの例です．プログラムを走らせた例をあげます．

```
0: exit, 1: item with weight, other: Item => 3
Item, price  ? 本 1500
0: exit, 1: item with weight, other: Item => 3
Item, price  ? 昼食 1000
0: exit, 1: item with weight, other: Item => 1
Item, price, weight [kg] ? => 米 3000 5
```

```
0: exit, 1: item with weight, other: Item => 1
Item, price, weight [kg] ? => 豚肉 500 0.2
0: exit, 1: item with weight, other: Item => 0
item    price   weight[kg]
本      1500
昼食    1000
米      3000    5
豚肉    500     0.2
------------------------------
total:  6000
```

この家計簿プログラムでは，物品の名前と価格，場合によっては重さも入力します．項目（Itemクラス）と，それに重さを加えたクラス（ItemWeight）を実装しています．ItemWeightはItemを継承していて，共通の部分のコーディングを無駄にコーディングはしていません．重さ情報のある項目にするかどうかは，入力時に判断します．家計簿でリスト状にvectorに保存していて，1つのリストに入れるために，各要素は基底クラスのポインタで，入力時にItem，ItemWeightどちらを実体化するかを判断します（ポリモーフィズム）．入力終了すると，リストと合計金額を表示します．

8.9 継承を使うのか使わないのか

継承は自然な発想だと感じる人が多いでしょう．動物の種類，自動車の種類，図形の種類，等々，基底クラスを継承した派生クラスは想像できると思います．ただ，基底クラスのデザインには，派生クラスで必要な機能まで想定している必要があります．基底クラスの変更は派生クラス全てに影響を及ぼすので，変更するには気を使う必要があります．また，文法も単純ではないことが実感できたでしょう．

継承について，自分でコードを書いて勉強することは大いに勧めます．一方，実用を求める際は，8.1節でも説明したように，必要と感じない場合には，自分から積極的に使うことはあまり勧めません．クラスを拡張する場合には，新しいクラスのメンバに元のクラスのオブジェクトを含めること（“Have a”の関係）で実装できるのであれば，その方が普通は楽です．コーディングでは楽をできるところで，楽をするのは良いことです．

以下に継承を使った場合と使わない場合の具体例を見てみます．まず，抽象クラスを使って継承したコード例です．

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

class Animal{
public:
  Animal() : weight_(0){}
  virtual void naku() =  0;
  virtual ~Animal(){}
  void setWeight(double w){ weight_ = w;}
private:
  double weight_;
};

class Dog: public Animal{  // 継承 "Is a"
public:
  void naku() override { cout<<"bow wow"<<endl; }
};

class Cat: public Animal{ // 継承 "Is a"
public:
```

```
  void naku() override{ cout<<"meow"<<endl; }
};

int main(){
  Dog dog;
  Cat cat;
  dog.naku();
  cat.naku();
}
```

継承を用いないコード例です．

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

class Animal{
public:
  Animal() : weight_(0){}
  void naku(string s) const{ cout<<s<<endl; }
  void setWeight(double w){ weight_ = w;}
private:
  double weight_;
};

class Dog{
public:
  void naku() const { anim_.naku("bow wow"); }
private:
  Animal anim_; // "Have a"
};

class Cat{
public:
  void naku() const { anim_.naku("meow"); }
private:
  Animal anim_; // "Have a"
};

int main(){
  Dog dog;
  Cat cat;
  dog.naku();
  cat.naku();
}
```

上の 2 つのプログラムの作動は同じで，実行すると次の出力をします．

```
bow wow
meow
```

どちらの場合にも動物（Animal）に犬（Dog）と猫（Cat）がいて，鳴き声が違います．main内は全く同じです．機能，特性を色々加える事を想定しています．使ってはいませんが，どちらの場合にも体重weight_とそれをセットするメンバ関数は実装しました．抽象クラスではオーバーライドすることは想定していないのでvirtual関数にしていません．使い方は実装していません．どちらの場合も，デフォル

トではweight_を0に初期化しました．継承を使わなくても，機能的には同等に整備できることが想像できるでしょう．

継承が必要となる場面もあるのには注意が必要です．1つは，ポリモーフィズムを使う場合です．上の例でも，動物の配列を作る場合には継承を使う必要があります．前節の家計簿のコード例でも1つ1つのアイテムの機能であれば，継承を使わず実装できますが，アイテムの配列を作るために継承を使っています．このような場合以外でも，GUIアプリケーションのプログラミング等で，現代的なプラットフォームを活用する際には，継承を使う必要があります．ただし，この場合は派生クラスをコーディングはしますが，基底クラスはコーディングしないで良い場合がほとんどです．

上で見てきたように，基底クラスは継承されることを想定してデザインすべきです．基底クラスとすることを想定していないクラスを継承して使ってしまう，あるいは継承されて使ってしまわれる事態は避けるべきです．以下のfinal宣言をすることで，クラスが継承されることを禁止できます．

final宣言：継承禁止

```
class クラス名 final {
  メンバ定義
};
```

単にクラス名の後にfinalを加えるだけです．final宣言したクラスを継承しようとすると，コンパイルエラーになります．

まとめ：継承

- 派生クラス：基底クラスに機能を付け加えられる．
- 基底クラスのpublic，protectedメンバを派生クラスでアクセスできて，それぞれpublic，protectedメンバになる（public継承）．
- 基底クラスのポインタは派生クラスのオブジェクトを指せる．
- 派生クラスでオーバーライドする可能性がある基底クラスのメンバ関数はvirtual関数にする（特に，基底クラスのデストラクタは常にvirtual）．
- 抽象クラス：pure virtual関数を持つクラス．実体化できない．

Chapter 9

様々な文法のポイント

本章では，ここまでで説明しきれなかったいくつかの重要であったり，知っておくと便利な文法のポイントについてまとめます．

9.1 テンプレート

9.1.1 テンプレートとは

```
#include <iostream>
#include <complex> //複素数ヘッダ
using namespace std;

template <class T>  //一般の型T
T sqr(T x){ return x*x; }  //2乗を計算

int main(){
  const int n(3);
  cout<<n<<"^2:\t"<<sqr(n)<<endl; //int引数で関数呼ぶ
  const double x(0.1);
  cout<<x<<"^2:\t"<<sqr(x)<<endl;//double引数で関数呼ぶ
  const complex<double> z(1,2);
  cout<<z<<"^2:\t"<<sqr(z)<<endl;//complex引数で関数呼ぶ
}
```

テンプレートは様々なデータの型を各型についてではなく，一般的に扱う方法です．上の例はテンプレートを用いて様々な型のデータの2乗を計算するプログラムで，実行すると次の出力を得ます．

```
3^2:    9
0.1^2:  0.01
(1,2)^2:        (-3,4)
```

整数の2乗，$3^2 = 9$，実数の2乗，$0.1^2 = 0.01$，複素数の2乗，$(1 + 2i)^2 = -3 + 4i$を計算しています．一般の引数の型Tの変数を2乗する関数を定義しています．Tは，いわば型の変数で，一般の型について関数を定義できます．1つの操作であれば，関数を 1 つ定義するだけで十分であるべきで，テンプレートを使えば実際そうなります．テンプレートを使わないと，各型について実質同じ関数を定義する必要が出てきます．テンプレートの便利さは明らかでしょう．

このsqr関数はTがどの型でも使えるのでしょうか？関数内の指示（この場合は掛け算）が定義されていれば使えます．たとえば，上のsqr関数はこのままではstringには使えませんが，stringについても*演算子を定義すれば使えます（9.3節参照，掛け算を定義したいかどうかは別問題です）．また，Tのかわりに Xのように他の変数名を用いても良いです． 1 文字である必要も，大文字である必要もありません．テ

ンプレート1個であれば（テンプレート頭文字）Tを使う場合が多いです．小文字1文字は，通常の変数と見間違いやすいので避けた方が良いでしょう．

テンプレート関数宣言，定義（型T）

```
template <class T>
戻し値型 関数名 (引数型 引数){ ...}
```

テンプレート関数宣言，定義（複数の型を用いる場合）

```
template <class T,class U,...>
戻し値型 関数名 (引数型 引数){ ...}
```

テンプレート関数宣言ではtemplate <typename T>のように，classのかわりにtypenameを用いても良いです．以下ではclassを用います．template <class T>と，型Tの変数も用いること以外は，通常の関数の定義と同じです．template <class T>直後の改行は必要ありません．ここではわかりやすいように含めました．

　複素数が初めて登場したので，触れておきます．複素数を使う場合は，#inclue <complex>で複素数関係のヘッダを含めます．complexもテンプレートを使用したクラスの例で，整数の複素数，実数の複素数をcomplex<int>，complex<double>で指定します．このような<>を用いた型の指定はvector（6.8節参照）で既に用いてるのでわかるでしょう．

9.1.2 テンプレートをメンバに用いたクラス

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

template <class T>
class A{
public:
  A(T a0): a(a0){} //コンストラクタ
  void print() const { cout<<"content: "<<a<<endl; }
private:
  T a;
};
int main(){
  A<int> aInt(1);
  aInt.print();
  A<string> aString("message string");
  aString.print();
}
```

プログラムの出力は以下のとおりです．

```
content: 1
content: message string
```

メンバデータaの型にテンプレートを使っているので，あらゆる型（自分で定義したクラスを含め）のデータをメンバにできます．関数の場合と同様，template <class T>と型Tの変数を用いる以外は，通常のクラスの宣言と同じです．内容を確認するメンバ関数printを含めました．vector,complex同様に<>内で変数の型を指定します．print関数は<<が定義されている型のデータであれば使えます．

　テンプレートを用いたクラスのメンバ関数をインラインではなくクラス宣言外で定義する場合はテンプレートを含める必要があるので，注意しましょう．テンプレートが無いとクラスではないからです．たとえば，上の例では，以下のようになります．

```
template <class T>
class A{
public:
  A(T a0): a(a0){} //コンストラクタ
  void print() const;
private:
  T a;
};

template <class T>
void A<T>::print() const { cout<<"content: "<<a<<endl; }
```

クラス宣言，定義以外の部分は同じなので省きました．

9.2 STL

9.2.1 STLとは

STLはStandard Template Library（標準テンプレートライブラリ）の略で，C++言語標準の一部です．STLにはコンテナとよばれる複数のデータをまとめたクラスと，それに作用する便利なツールが含まれています．コンテナの典型例はvector（6.8節参照）です．vectorの例からわかるように，一般の型のデータを要素とするためにテンプレートが使われています．コンテナには，vector以外にも値の重複していない集合（set），重複が許される集合（multi_set）等もあります．ここでは，vectorで典型的な使い方を見ていきます．

9.2.2 イテレータ（反復子）

一般にコンテナ，そしてそれに作用するツール類を扱うためには，**イテレータ（反復子）**の使い方を理解する必要があります．イテレータは，特定のコンテナに作用するポインタのようなものです．

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

int main(){
  vector<int> v  = {-1,0,1,10,100};
  for(vector<int>::iterator it = v.begin() ;
        it != v.end() ; ++it){
    cout<<*it<<" ";
  }
  cout<<endl;
}
```

上のプログラムの実行結果は以下のとおりです．

```
-1 0 1 10 100
```

vectorの初期化に{…**中身**…}を使いました（6.10節参照）．vector<int>に作用するイテレータの型はvector<int>::iteratorです．初めの要素のイテレータはbegin関数で得られ，最後の要素の次の要素（にあたるもの）のイテレータはendになります（図9.1参照）．++でイテレータを１つ進められます（逆に--で１つ戻せます）．イテレータには大小関係，>，< は使えません．上のコードでは全ての要素を出力するために，beginから，endの**１つ手前**までイテレータを動かしました．ポインタ同様に，*itがイテレータの指すデータになります．上の例では，*v.begin()の値が-1，*(v.begin()+2)の値は1になります．

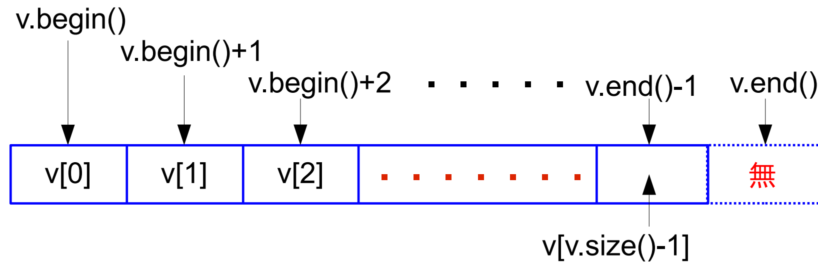


Figure 9.1: vectorのイテレータとインデックスで指定した要素の関係.

イテレータを使う際には，ポインタ同様に指す変数の中身を書き換えることもできる事を意識する必要があります（6.3節参照）．よって，上の例のように書き換えをしない場合は，明示的にiteratorの代わりにconst_iteratorを使う事を勧めます．そうすれば，書き換えようとするとコンパイルエラーになります．

*(v.begin())よりv[0]の方がわかりやすいのになぜイテレータなんか使うのだろうと感じる人もいるかも知れません．まず1つの理由は，vectorではインデックスを使って要素を呼び出せますが，一般のコンテナではそれができないことです．どのコンテナでも，イテレータを使えばデータにアクセスできます．もう1つの理由は，次節で説明するコンテナに作用する便利なツール類を使いこなすためにはイテレータが必要なことです．

9.2.3 algorithm

STLのコンテナに作用する便利なツールがalgorithm（アルゴリズム）とよぶライブラリにまとめられています．いくつか例をみてみましょう．使うためには，#include <algorithm>でヘッダを含める必要があります．

sort

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

template <class T>
void printVector(const vector<T> &v){ //template, reference
  for(unsigned int j=0 ; j<v.size() ; ++j){
    cout<<v[j]<<" ";
  }
  cout<<endl;
}

int main(){
  srand(0);
  vector<int> v;
  for(vector<int>::size_type  j=0 ; j<10 ; ++j){
    v.push_back(rand()%100);
  }
  printVector(v);
  sort(v.begin(),v.end());
  printVector(v);
}
```

プログラムの実行結果は次のとおりです．

```
83 86 77 15 93 35 86 92 49 21
15 21 35 49 77 83 86 86 92 93
```

コードでは，2桁の乱数10個をvectorに入れ，それを小さい順からに並べ替え（sort）ています．並べ替え前後でデータ出力をしています．並べ替えができていることがわかるでしょう．並べ替えはsort(v.begin(),v.end());だけでできています．ソーティング（並べ替え）が標準的に装備されていて，単にsortを呼ぶだけで並べ替えができるのは便利です．上ではsortでbeginからendで範囲を指定し，全部並べ替えましたが，一部のデータの並べ替えもできます．たとえば，3個目から8個目までの6個を並べ替える場合は，sort(v.begin()+2,v.begin()+8);と書けます．他も同様です．再現できるように乱数をsrand(0);で初期化しています（5.8.1節参照）．

sortの使い方

```
sort(ソートする初めの要素を指すイテレータ,
    ソートする最後の要素の次を指すイテレータ);
```

上のコードではvectorを標準出力に出すための関数，printVectorを含めました．vectorの要素の型にテンプレートを用いています．ループ変数は0以上なのでunsigned intを使っています（2.3.2節参照）.[1]不要なコピーを作らないために，const宣言とともに参照を用いていることにも注意しましょう．

max,min_element

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

int main(){
  srand(0);
  vector<int> v;
  for(vector<int>::size_type  j=0 ; j<10 ; ++j){
    v.push_back(rand()%100);
  }
  printVector(v);
  cout<<"Max: "<<*max_element(v.begin(),v.end())<<endl;
  cout<<"Min: "<<*min_element(v.begin(),v.end())<<endl;
}
```

プログラムの実行結果は以下のとおりです．

```
83 86 77 15 93 35 86 92 49 21
Max: 93
Min: 15
```

2桁の乱数10個をvectorに格納し，その最大値と最小値を求めるコードです．一目瞭然だと思いますが，max_element，min_elementの戻り値がイテレータであることに注意が必要です（よって*を前に付けています）．printVectorは前節と同じコードなので省略しました．

algorithmには，他にも数多くの便利な関数が用意されているので，何があるか目を通しておくと良いでしょう．上で扱った以外にも，範囲内のデータに同じ操作をする，中で検索する，条件を満たす要素を数える，コピーする，逆に並べる，シャッフルする，合成する，等の様々な関数があります．

[1] size_typeを使う場合は，forの部分を for(typename vector<T>::size_type j=0 ; j<v.size() ; ++j)と書き換えます（6.8.1節参照）．同様にfor(typename vector<T>::const_iterator it=v.begin() ; it != v.end() ; ++it)とすればイテレータを使えます．これらの場合には，型の宣言であることを明示するためにtypenameが必要です．

9.3 演算子のオーバーロード

9.3.1 演算子のオーバーロードとは

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

template <class T>
vector<T> operator+=(vector<T> &v1,vector<T> const &v2){
  if( v1.size() == v2.size() ){
    for( unsigned int j=0 ; j<v1.size() ; j++ ){
      v1[j] += v2[j];
    }
  }
  else{
    cerr<< "v1.size="<<v1.size()<<" != "<<v2.size() <<"=v2.size (+=)"<<endl;
    terminate();
  }
  return v1;
}
template <class T>
vector<T> operator+(vector<T> const &v1,vector<T> const &v2){
  vector<T> v(v1);
  return v += v2;
}

int main(){
  vector<int> v1({1,2}),v2({3,4});
  v1 += v2;
  for(unsigned int j=0 ; j<v1.size() ; ++j){
    cout <<v1[j]<<" ";
  }
  cout<<endl;
}
```

演算子のオーバーロードは，演算子の作用が定義されていないデータの型について作用を定義することです．上のコードはvectorについて演算子+を**オーバーロード**した例です．演算子+の作用はint，double等には定義されていますが，vectorには定義されていません．プログラム実行結果は次のとおりです．

```
4 6
```

ベクトルの足し算$(1,2)+(3,4)=(4,6)$の計算結果です．+を定義する際には，+=も同時に定義すべきです．まず+=を定義し，それを用いて+を定義しています．また，足す2つのvectorの次元が同じでないと定義できないので，チェックをし，同じでない場合はプログラムをterminate();で強制終了する仕様になっています．

演算子のオーバーロードの定義の仕方では，operatorの後に演算子を書き，それ以外は基本的に関数の定義と同じです．演算子はデータ変数に対して配置する位置が関数とは異なる「関数」とみなせます．たとえば，中身はそのままでoperator +=をvectorAddPlusに置き換えれば，vectorAddPlus(v1,v2)のように使える関数になります．

+を定義しないで，addVector(v1,v2)のような関数を定義することもできます．その場合は，3つvectorの足し算はaddVector(v1,addVector(v2,v3))のようになります．演算子オーバーロードを用いると，v1+v2+v3と書けるので楽なだけではなく，直観的にわかりやすくなります．これは間違いを避けるために有用です．

9.3.2 テンプレートを用いた演算子オーバーロード

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

template <class T>
ostream &operator<<(ostream &stream,vector<T> const v){
  for(unsigned int j=0 ; j<v.size() ; ++j){
    stream << v[j] << " " ;
  }
  return stream;
}
int main(){
  vector<int> v;
  for(int j=0 ; j<10 ; ++j){
    v.push_back(j);
  }
  cout<<v<<endl;
}
```

オペレータオーバーローディングを用いて，vectorを<<で出力できるようにしたのが上のコードです．プログラム出力は以下のとおりです．

```
0 1 2 3 4 5 6 7 8 9
```

<<はostreamに文字列を付け足し，さらに付け加えられるように戻します．ostreamを変化させるので，参照渡しを用い，さらに参照を戻しています（6.11節参照）．テンプレートを使っていますが，演算子オーバーロードの定義の仕方は同じです．

この演算子オーバーロードを使うと，vectorをprintVectorのような関数を使わずとも，int，double同様に出力できるので，便利なことがわかるでしょう．一般にvectorの中身を出力しようとすると，中身の型をどう指定するのかと悩むかも知れませんが，テンプレートを用いると上のようにあらゆる型について同じように扱えます．

9.3.3 演算子オーバーロードに関する重要な点

+,-,*,/,%,<<,>>,>,<を含むあらゆる演算子はオーバーロードできます．上の例から，演算子オーバーロードはコードを直観的にわかりやすくし，便利なツールとなりうる強力かつ魅力的な道具であることがわかるでしょう．しかし，気をつけなければならない重要なポイントがるので，説明します．

- **間違えない！** 演算子は直観的に意味がわかる（と感じる）ものであるので，間違えると，コードの間違い探しが大変困難になります．何でも間違えてはいけないのは当然ですが，特に演算子オーバーロードは間違えてはいけません．

- **必要な場合しか使わない！** 間違えると大変なことになるので，使用は必要な場合に限るべきです．気楽に使うものではありません．

- **自然な作用にする！** 演算子は直観的にその作用がわかるべきです．非直観的なふるまいをすると，コードを読んでも意味がわからなくなります．極端な例をあげると，**文法的観点だけからであれば，** vectorの引き算を（-ではなく）+演算子の作用として定義できます．ただ，このような事をすると，コードの作用が全くわからなくなりかねないので，絶対にしてはいけません．何が「自然」かという微妙な問題もあります．たとえば，vectorの+を通常のベクトルの足し算として上では解釈しましたが，集合的に考えて$(1,2)+(3,4)=(1,2,3,4)$の方が「自然」という解釈もありえます．こういった点にも留意すべきです．

9.4 mainとargc，argv

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

int main(int argc, char *argv[]){
  cout<<"argc: "<<argc<<endl;
  for(int j=0 ; j<argc ; ++j){
    cout<<argv[j]<<endl;
  }
}
```

main関数では引数を用いることにより，main関数がどのように呼びだされたかの情報を得られます．上のプログラム（実行型ファイルをa.outとします）を走らせると，以下のように出力されます．

```
> a.out this is a test
argc: 5
./a.out
this
is
a
test
```

関数を呼び出した際の実行型ファイル名と引数が配列（argv）に入っていて，この配列の要素数がargcです．配列の初めの要素は呼び出したプログラムのファイル名なので，配列argvの要素数は引数よりも1多いことに注意しましょう．変数名をargc（argument count，引数の数），argv（argument vector，引数配列）とするのが慣習ですが，変数名は何でも良いです．

今までは，プログラムを走らせ始め，それから入力を求める仕様のコードを書いてきましたが，上のように書くと呼び出す際に入力できます．どちらの仕様を選ぶかは通常は好みの問題ですが，片方が必要な状況もあります．元来，C言語はOSを書くために作られたプログラミング言語です．OSではたとえば以下のようなコマンドが用意されています．

```
$ ls -al /home/god
```

コマンドラインでオプション，ファイル名等をlsの引数として与えます．この操作の実装にはargc，argvを用います．

9.5 ブロックとスコープ

9.5.1 ブロック，スコープとは

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

int main(){
  { //ブロック
    int n = 1;
    cout<<n<<endl;
  }
  // cout<<n<<endl; //コンパイルエラー
}
```

基本的に，あらゆるデータ名はある**ブロック**内で定義されています．ブロックは{}でくくられた指示のまとまりです（5.1節参照）．上のコードでは，nはブロック内でしか定義されていません．よって，ブロックの外で使おうとすると，コメントアウトした行のようにコンパイルエラーになります．変数の定義されている範囲を変数の**スコープ**とよびます．ブロックの外では，nのスコープから外れているのです．

上のブロックとコメントした部分の{}を無くすと，nはmain関数全体で使えます．つまり，main関数ブロックがスコープとなります．一般に，関数の定義は{}内に指示を書きますが，これが関数ブロックです．forループでも指示をfor(...)に続く{}内に書きますが，これもブロックです．

クラスを使った例を見てみましょう．

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

class A{
public:
  A() { cout<<"コンストラクタ"<<endl; }
  ~A(){ cout<<"デストラクタ"<<endl; }
};
int main(){
  cout<<"スコープ前"<<endl;
  {//ブロック
    A a;
  }
  cout<<"スコープ後"<<endl;
}
```

プログラムの出力結果は次のとおりです．

```
スコープ前
コンストラクタ
デストラクタ
スコープ後
```

ブロックに入って，クラスAのオブジェクトaが実体化され，ブロックを抜けるとスコープから外れて，自動的に廃棄されていることがわかるでしょう．

9.5.2 グローバルデータ

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

int n = 1; //グローバル変数定義

int main(){
    cout<<n<<endl;
}
```

{}外で変数を定義したのが上の例です．この場合のnのスコープは宣言以降の**ファイル**になります．よって，main内でnを定義していなくても使えます．このようなデータを**グローバルデータ**，**グローバル変数**とよびます．

次の例も見てみましょう．

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

```
int n = 1; //グローバル変数定義

int f(int p){ return p+n; }

int main(){
  cout<<f(0)<<endl;
}
```

$0+1=1$なので，プログラム出力は次のとおりです.

```
1
```

関数f内では，明示的にはnは定義していないのに，スコープ内なので使えます.

初めに定義すれば，ファイル内どこでも使えるのでグローバルデータは便利だと思うかも知れません. しかし，どこでも使えると，意図しない間違いを起こしやすくなり，また，間違いを起こした際に箇所を特定するのも難しくなります. グローバルデータは特に理由が無い限り使うべきではありません.

グローバルデータを使うのが適切な場合はあるのでしょうか？たとえば，全体で共通な定数を定義するのは自然な使い方の例です. 自分で定数を定義する際はconstを使い，意味が明らかなように定義しましょう. たとえば，真空中光速を以下のようにグローバル変数として定義しても良いでしょう.

```
const double LIGHTSPEED = 2.99792458e8; //[m/s]真空中光速
```

ここで，e8というのは10^8の意味です. たとえば，上の値は2.99792458×10^8です. 全て大文字にし，定数であることがすぐにわかるようにしています.

9.5.3 スコープに対する考え方

基本的に**データのスコープはできるだけ狭くすべきです**. ループ内変数のスコープはループブロックにすべきだ（5.6.1節参照），というのもその例です. そうすれば，意図しない箇所でそのデータが用いられる事が少なくなり，安全だからです. さらに，間違いを探す際にもデータの存在している範囲が狭い方が間違いを見つけやすいです. この点については，グローバルデータを説明する際にも指摘しました.

ただ，スコープを厳密に狭くしていこうとすると，それぞれのデータの使われる範囲を特定し，{}でくくっていく必要があります. これはあまり現実的ではありません. よって，常識的な範囲で収めると良いでしょう. たとえば，ループ変数はループブロックがスコープ，ifブロック内でしか必要ない変数はifブロックがスコープ，グローバルデータは必要無い限り避ける，等々です.

デバッグする際には，どこでどのデータが使われているかの情報であるスコープの概念は重要です. コンパイルはするけど誤動作するコードでは，意図的に特定のデータのスコープ（と考えている部分）を{}でくくってコンパイルしてみると良い場合があります. コンパイルエラーが出た場合は，意図しない箇所で使っていることがわかります.

9.6 自動変換

9.6.1 自動変換とは

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

int main(){
  double x = 3.9;
  int n = x; //自動変換
  string s = "Hello!"; //自動変換
  cout<<x<<", "<<n<<", "<<s<<endl;
```

```
}
```

上のプログラムの出力は以下のとおりです.

```
3.9, 3, Hello!
```

よく見ると，int型変数nにdouble型変数xの値が代入されています．型が一致していませんが，double型の変数の値が**自動変換**されてint型になっています．2.6節で説明したように，整数にする場合は切り捨てられていることに注意しましょう．また，上の例ではsに定数文字列を代入していますが，これがC++のstringに自動変換されています．

上の例とは逆に，

```
double x = 3;
```

と書くと，整数の3がdouble型に自動変換されています．

関数の引数でも自動変換が起きます．

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

double f(double x){ return 5*x; }
string g(string s){ return "string is: "+s; }

int main(){
  cout<<f(10)<<endl;
  cout<<g("C++ is great!")<<endl;
}
```

プログラムの出力は以下のとおりです．

```
50
string is: C++ is great!
```

関数fでは引数に整数を代入していますが，double型に自動変換されています．計算5*xで整数5もdouble型に自動変換されています．関数内では変数が一旦コピーされてから使われる（6.11節参照）ので，直前のコード例の=で説明した効果と同じだと考えればわかりやすいでしょう．

上のコードを見ても，何も新しい事が無いように感じたかも知れません．今までも出てきていますが，気付いていないかも知れません．通常は自動変換はそれほど自然なものです．逆に，自動変換が無いと，コーディングで大変不便です．一方，不自然な自動変換を許すとコードがわかりにくくなりえます．これを次に考えてみましょう．

9.6.2 自動変換で気を付けるべき点

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

class A{  //クラスAの定義始まり
public:
  A(int n0) : n(n0) { } //引数有りコンストラクタ
  int getN() const { return n; }  //ゲッター
private:
  int n;
};        //クラスAの定義終わり

int main(){
  A a = 200; //自動変換
```

```
  cout << a.getN() <<endl;
}
```

自動変換はいいことづくめのように見えますが，コードが分かりくくなる場合もあります．上では，4.4節のクラスをちょっと改変したものを用いました．プログラムの出力は以下のとおりです．

```
200
```

オブジェクト実体化 A a = 200; が当たり前に見えるでしょうか？200がコンストラクタで，オブジェクトに自動変換されています．A型の引数を持つ関数に数字を入れても自動変換されます．文法的には正しいですが，このような自動変換はコードを読みにくくして危険もあります．このような自動変換が意図せず起きないように，禁止できます．上のコードで

explicit A(int n0) : n(n0) { }

のようにexplcitをコンストラクタ宣言に加えれば良いだけです．こうすると，オブジェクトを実体化するには，明示的に（explcitに），たとえば

A a(200);

と指示する必要があります．安全で手間も大して増えません．よって，原則として引数が 1 個のコンストラクタの宣言にはexplicitを付けることを勧めます．引数が2個以上の場合は自動変換しないので，考慮する必要はありません．

9.7 コンパイラ，プリプロセッサ，ヘッダファイル

9.7.1 #で始まる行とプリプロセッサ

#include <iostream> のように#で始まる行は，C++コンパイラで処理される前に，プリプロセッサが処理します．これがプリプロセッサの働きです．プリプロセッサへの指示は，全て行が#で始まり，行の終わりまでです（;はいりません）．#の前の空白は許されますが，行の1字目を通常#にします．行の終わりに意味があることからわかるように，C++の指示とは異なり，フリーフォーマットではありません．プリプロセッサがどのような処理をするのか，典型的なプリプロセッサの指示から理解しましょう．

9.7.2 #include

今まで使ってきたプリプロセッサへの指示は#includeだけです．これは，単に#includeの後のファイルを読み込むだけです．プリプロセッサの処理だけをしてみるとわかります．[2]ここまでは，#includeを用いて，既に準備されている標準のヘッダファイルを読み込む場合のみを扱ってきました．

#includeまもっと一般的で，自分で書いたヘッダファイルを含め他のヘッダファイルも読み込めます．次の例を見てみましょう．ファイル名func3.hppのファイルを作ります．

```
// ヘッダファイル func3.hpp
#include <iostream>

int func3x(int); //関数宣言
```

C++のヘッダファイルの拡張子は.hppか.hにします．次のコードを上のヘッダファイルと同じディレクトリ（フォルダ）に作ります．

```
#include "func3.hpp"
using namespace std;

int main(){
  cout << func3x(10) << endl; // 関数をよぶ
```

[2] g++ (GNUプロジェクトの C++コンパイラ)やicc (インテル社のC++コンパイラ）では，“gcc -E”，“icc -E”，のように“-E”のオプションを付けることで，プリプロセッサだけの処理結果を確認できます．

```
}

int func3x(int x){  // 関数定義
  return 3*x;
}
```

上で説明したように，#includeは単にfunc3.hppを読み込むだけなので，これは，4.2節のコード例と実質同じになり，同じ作動をします．この例では，1つのファイルを2つにしていて，何も役に立たないように見えるかも知れません．便利になる場合をあげます．たとえば，同じ関数を様々なプロジェクトで用いる場合に，関数（上ではfunc3）を別ファイルにして，ヘッダと関数を定義したファイルを共有できます．また，たくさん要素がある場合には1つのファイルにまとめず，複数のファイルに分けます．分けたファイルでクラスや関数の宣言を共有するためにヘッダファイルを使います．ファイルを複数に分けた場合には，コンパイルしたあとに全部をまとめてリンクする必要があります（リンクの仕方は環境によって異なりますが，IDE（付録A.0.2参照）を用いている場合は自動的に行われます）.

9.7.3 #define

#include <cmath>で読み込まれる，標準のヘッダファイルには次の記述があります.

```
# define M_PI           3.14159265358979323846  /* pi */
```

これはコードでM_PIを次の数字で置き換える，というプリプロセッサへの指示で，コンパイル時には数字に置き換わっています．よって，次のコードをコンパイルして実行すると

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
  cout<<M_PI<<endl;
}
```

次の出力を得ます（デフォルトでは数桁しか表示されません).

```
3.14159
```

ここで，注意して欲しいのはあくまでも**置換**であることです．この機能を用いて，関数を定義することもできます.

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

#define f1(x,y) ((x)+2*(y))

int main(){
  cout<<f1(1,2)<<endl;
}
```

上のコードをコンパイルして実行すると次の出力を得ます.

```
5
```

$1+2*2=5$と理解できます．これを見ると，引数の型を指定しないで，簡潔に関数が定義できて便利だ！と感じるかも知れません．結論からいうと，C++では上のような関数（マクロ関数ともよびます）の定義は勧めません．主な理由をあげます.

- 上の定義では()が多いことに気づいたかも知れません．これは， #defineは単なる置換なので必要です．#define f1(x,y) (x+2*y)と定義すると，f1(a,b+c)は(a+2*b+c)と置換されます．(a+2*(b+c))になりません．置換だからです．また，f1と(x,y)の間に空白を入れると，f1をそれ以降(x,y) ((x)+2*(y))と置換してしまいます！このように，間違いを犯しやすいです．普通に，C++の関数として定義した方が間違いをしにくくて安全です．
- x,yの型を指定しないのは楽で良いように思うかも知れませんが，C++ではテンプレートを用いて一般の型について無駄な手間なく関数を定義できます．さらに，テンプレートを含め，型を指定する方がコンパイル時のチエックが働き，間違いをしにくいので安全です．

定数の定義に#defineを使うのは問題無いですが，その場合も，const int N=100;のように，const宣言を伴い型を指定した定数の方が安全なので，そちらの方を勧めます．

9.7.4 #if，#ifdef，#ifndef，#elif，#else，#endif，defined

プリプロセッサにも，#if，#elseといった指示ができます．ここでは，いくつかの便利な使い方の例だけ示します．

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

#define INT64
#ifdef INT64
# define INT int64_t
#else
# define INT int
#endif

int main(){
  INT n;
  cout<<"Integer => ";
  cin>>n;
  cout<<n*n<<endl;
}
```

上のコードを実行し，200000⏎と入力してみます．

```
Integer => 200000
40000000000
```

上のコードでは，INT64が定義されて（defined）いれば，INTがint64_t（64ビット整数型）になり，そうでなければINTがint（整数型）になります．INT64が特定の値に定義されている必要はありません．#ifdefは#if definedと同義で，#ifでの条件分けは#endifで終える必要があります．#define INT64の行をなくすと，同じ入力で出力は次になります．

```
Integer => 200000
1345294336
```

整数が溢れてしまっています．同じコードを環境によって（使えるメモリ，要求される精度，等々）によって，変化させるためにプリプロセッサの指示を使えます．定義（この場合はINT64）はコードに書かないで，コンパイラのオプションでも指定できます． OS，コンパイラ，様々なアプリケーションのコードは，環境によって，コンパイルするコードが自動的に変更されるように#ifdef等は多く使われています．興味があったら，調べてみて下さい．

これ以上はプリプロセッサへの指示の仕方は詳しくは説明しませんが，!,||,&&なども使えます．#ifndef XはXが定義されていない場合に真で，#if !(defined X)と同値です．

includeガード

```
#ifndef FUNC3HPP
#define FUNC3HPP
#include <iostream>

int func3x(int); //関数宣言
#endif
```

便利でよく使われるプリプロセッサの用法を説明します．上のコードは本節で登場したfunc3.hと実質同じです．プリプロセッサの指示を用いて複数回読み込むことを防止しています．立て続けに2度でも3度でも読み込んでみて下さい．エラーになりません．仮に関数の定義int func3x(int);を2度しようとしたらコンパイル時のエラーになります．このコードでは1度目に読み込むと，FUNC3HPPが定義されていないので，FUNC3HPPを定義するとともに，func3x関数の定義を読み込みます．2度目に読みこもうとすると，FUNC3HPPが定義されているので，#ifndefが偽になり，func3x関数の定義は読み込まれません．よって，複数回定義しようとするエラーを防止しています．この仕組みをincludeガードとよびます．

同じヘッダファイルを複数回読み込もうとする間違いをすることは無いだろうと思うかも知れません．しかし，これは意外と多く生じます．ヘッダファイルでは，他のヘッダファイルを読み込む事が多いです．上の例でもiostreamを読み込んでいます．複数のヘッダファイルを読み込むと，両方でiostreamを読み込むかも知れません．また，読み込むヘッダファイルに#includeされているヘッダファイルを再度読み込もうとするかも知れません．このような場合でも，includeガードはエラーから守ってくれるので，便利です．実際，標準のヘッダファイルには全てincludeガードが使われているので確認してみてください．

9.8 関数オブジェクト（ファンクタ）

9.8.1 関数オブジェクトとは

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

class FuncPow{ // 関数オブジェクト
public:
  explicit FuncPow(double p): p_(p){}
  double operator () (double x) { return pow(x,p_); }
private:
  double p_;
};

int main(){
  FuncPow f2(2);
  FuncPow f0p1(0.1);
  for(int j=0 ; j<5 ; ++j){
    const double x(0.1*j);
    cout<<x<<"\t" << f2(x)<<"\t"<<f0p1(x)<<endl;
  }
}
```

関数オブジェクト（あるいは**ファンクタ**)は関数のように使えるオブジェクトです．上のプログラムを走らせると以下の出力を得ます．

```
0       0       0
0.1     0.01    0.794328
0.2     0.04    0.85134
```

```
0.3     0.09    0.886568
0.4     0.16    0.912444
```

上のプログラムでは，関数x^pを計算する関数オブジェクトのクラスを定義し，それを$p = 2, 0.1$の場合の関数オブジェクトf2，f0p1を実体化しています．これを用いて$x = 0, 0.1, 0.2, 0.3, 0.4$について関数の値を表示しています．その際，メンバ関数を使わずに普通の関数のようにf2(x)，f0p1(x)で値を求めているのが，関数オブジェクトとよぶ理由です．オブジェクトであるので，メンバデータ（この場合はp_）が使えることに注意しましょう．

文法を見てみましょう．FuncPowクラスは通常のクラスの定義で，operator···で()をオーバーローディングしているだけです（9.3節参照）．よって，基本的な考え方は今までに学んだものです．FuncPowではコンストラクタで指数pを初期化するようにしました．ただ，引数が1個のコンストラクタなのでexplicitのキーワードを安全のためにコンストラクタ宣言に含めています（9.6節参照）．

関数オブジェクトにおける()のオーバーロード

戻り値型 operator ()（引数）{ 関数内の指示; }

9.8.2 関数オブジェクト：応用

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

class FuncPow{
public:
  explicit FuncPow(double p): p_(p){}
  double operator () (double x) { return pow(x,p_); }
private:
  double p_;
};

class FuncExp{
public:
  double operator () (double x) { return exp(x); }
};

template <class F>
class  IntSimpson{
public:
  IntSimpson(const F &f,int n=1000): fObj_(f),nInteg_(n){
   nInteg_ += max(( nInteg_%2 ? 1 : 0 ),100); }
  double integrate(double,double);
private:
  F fObj_;
  int nInteg_;
};

template <class F>
double IntSimpson<F>::integrate(double x0,double x1){
  double sum(fObj_(x0)+fObj_(x1));
  const double dx((x1-x0)/nInteg_);
  for(int j=1 ; j<nInteg_; ++j){
```

```
      const long double x(x0+j*dx);
      sum += ( j%2 ? 4 : 2 ) * fObj_(x);
  }
  return sum*dx/3;
}

int main(){
  cout<<"---- x^a ---"<<endl;
  for(int j=0 ; j<5 ; ++j){
    const double p(0.6*j);
    FuncPow fPow(p);
    IntSimpson<FuncPow>  simpPow(fPow);
    const double result(simpPow.integrate(0,1));
    cout << p<<":\t"<<result<<"\t"<<result-1/(p+1)<<endl;
  }
  cout<<"---- exponential ---"<<endl;
  FuncExp fExp;
  IntSimpson<FuncExp>  simpExp(fExp);
    const double result(simpExp.integrate(0,1));
    cout << result<<"\t"<<result-(exp(1)-1)<<endl;
}
```

上の例は積分$\int_0^1 x^p\,dx = 1/(p+1)$，$\int_0^1 e^x\,dx = e-1$を数値積分で求め，正確な値からのずれをチェックするプログラムです．走らせると次の結果を得ます．eを含む数字は科学表記で，たとえば，-7.67032e-07は-7.67032×10^{-07}を意味します．

```
---- x^p ---
0:      1       0
0.6:    0.624999        -7.67032e-07
1.2:    0.454545        2.21243e-09
1.8:    0.357143        2.26608e-11
2.4:    0.294118        -7.63944e-13
---- exponential ---
1.71828 6.43929e-15
```

FuncPowクラスの定義は前節と同一で，指数関数の関数オブジェクトFuncExpの定義を加えました．テンプレートを用いて，doubleの引数と戻り値を持つ一般の関数オブジェクトについての数値積分を求めるクラス，IntSimpsonを定義しています（9.1節参照）．よって，FuncPow，FuncExpともに同じ数値積分クラスIntSimpsonで求めています． x^pを計算するのに，単に関数を

```
double f(double x,double p){ return pow(x,p); }
```

と定義して求めれば良いのになぜクラス定義など面倒なことをするのだろうかと思ったかも知れません．関数値を計算するだけであれば，確かに関数で十分です．ただ，上の例で考えてみましょう．積分はいわば関数の関数なので（汎関数とよびます），数値積分の引数は関数です．この場合，引数である関数の型（引数を含め）を指定しなければなりません．たとえば，上の例ではx^pの引数はx, p，e^xは引数がxだけ，と引数の数からして違います．よって，関数を数値積分に用いるには，それぞれの関数の型に特化したクラス（あるいは関数）を書くことが必要となってしまいます．両者を同じクラスで処理できる関数オブジェクトを用いた上の方法に利点があることがわかるでしょう．なお，テンプレートを用いたので，FuncPow，FuncExpに対応できたことにも注意しましょう．

関数オブジェクトではなく，関数を用いて数値積分クラス（あるいは関数）をコーディングする場合，関数の型を統一する必要が出てきます．どのような方法で実装できるか考えてみましょう．一般に関数には，積分する変数（ここではx）とパラメータ（p等）があり，パラメータの数は任意です．

1. 関数はf(x)とし，グローバル変数を用いる．

2. パラメータの数がわからないので，とりあえず，必ず関数は`f(x,p)`とし，`p`は実数の`vector`とする.

解決策1は単純で，短いコードであれば問題無いかも知れません．しかし，グローバル変数を用いると，それがどこで定義，変更されているのか等がわかりにくくなります．少しでも長くなったり，複数のファイルにまたがる場合等，単純なコード以外では避けた方が良いです．解決策2は実用的かも知れませんが，直観的ではなく，コーディングがしにくいです．さらに，パラメータの種類が実数ではない場合に対応できません．たとえば，関数がA, B, Cみたいなキーでふるまいが異なる場合も考えられます．このような場合にも対応するために，パラメータの一般的なクラスを引数に含める場合もあります．これらと比較して，上で説明した関数オブジェクトを用いた方法は，データの型も明示的で，1つのスマートな解決法だと考えています.

ここで用いた数値積分法はシンプソン公式とよばれる方法で，次のような近似です.

$$\int_a^b f(x)\,dx \simeq \frac{h}{3}\left[f(a)+4f(a+h)+2f(a+2h)+4f(a+3h)+\cdots+2f(b-2h)+4f(b-h)+f(b)\right]$$

積分領域を幅$h=(b-a)/N$のN区間に分けて積分を計算しています．単純な足し算に比較して，シンプソン公式は一般にオーダーh^4の誤差を生じるので，単純な足し算と手間がほとんど変わらないのに精度が高い便利な数値積分法です．上のコードではデフォルトパラメータを用いて，区間数`nInteg_`をデフォルトで1000としています．区間数は偶数の必要があるので,

```
nInteg\_ += max(( nInteg\_\%2 ? 1 : 0 ),100);
```

で奇数の場合は1加えて偶数にし，最低でも100としています．上のコードではデフォルト値を用いて$h=10^{-3}$なので，10^{-12}程度以内のずれが期待されます．本書ではこれ以上詳しくは書きませんが次数が$0.6, 1.2$の場合はずれが桁違いに期待より大きく，数値積分の厄介な部分が見えています．ちなみに，この場合は，正確な値がわかっているのに，なぜ数値積分をするのだろうと思うかも知れません．確かに上の関数であれば，数値積分をする必要はありません．しかし，一領域で上のようなふるまいをし，解析的に積分できない場合もあります．このような場合にどのように処理すべきか，という例，あるいは検証です.

9.8.3 なぜ関数オブジェクト

なぜ関数オブジェクトを使うのか，考えてみましょう.

- 関数のように使えるが，クラスなので，メンバデータを保持して用いる，メンバ関数を用いるなどのクラスの機能を使える利点がある．上の数値積分のコードもメンバデータの利便性が現れている例.
- 関数オブジェクトを用いれば単にオブジェクトを引数とするだけで，実質的に関数を引数にできる．関数自体を引数にすると，そのための異なる文法を使う必要がある.
- STLのalgorithm等で，多くの場合に関数オブジェクトを引数に使える．特にラムダ式を用いると便利な場合がある（具体例は次節）.

9.9 ラムダ式（無名関数）

9.9.1 ラムダ式とは

```
double f(double x,double p){ return pow(x,p); }
#include <iostream>
using namespace std;

int main(){
  int p(3);
  cout<<[=](int n){ return n/p; }(100)<<endl;
  auto f1  = [=](int n){ return p*n; };
```

```
  cout<<f1(5)<<"\t"<<f1(10)<<endl;
}
```

上のプログラムを実行すると次の出力を得ます．

```
33
15      30
```

上では**ラムダ式**（無名関数ともよびます）を用いており，これは無名の関数オブジェクト（前節参照）をその場で定義する方法です．ラムダ式の書き方は以下のとおりです．

ラムダ式

`[ キャプチャするデータ ](引数){ 関数内の指示; }`

引数と関数内の指示は普通の関数と変わりません．ラムダ式では**スコープ内のデータ**を用いることができます．**キャプチャ**するデータとは，関数内で使える（キャプチャする）データです．[=]はスコープ内の全てのデータをキャプチャします．[&]はスコープ内の全てのデータを参照してキャプチャします（6.3節参照）．[]であれば何もキャプチャしません．関数と比較した際に，名前が無い以外に，**戻り値の型の指定が無い**のが重要な特徴です．戻り値の型は自動的に決まります．

この文法を理解したうえで，コードを見ましょう．まず，

```
[=](int n){ return n/p; }(100)
```

はn/pを計算する関数で，それを引数100で呼んでいます．よって，33が出力されています．pはスコープ内の変数で，[=]でキャプチャしているので使えます．この場合は[p]，あるいは[&]，[&p]でも良いです．後者2つは参照を意味します．キャプチャ無しの[]ではコンパイルしません．

次のauto f1 = [=](int n) return p*n; ; ではラムダ式でp*nを求める関数オブジェクトを定義し，これに名前を付け，それを普通の関数のように用いています．ここではautoのキーワードを用いて関数オブジェクトの型を自動的に推定しています．autoについては次節でより詳しく説明しますが，autoを使わずに，具体的に方を書き下す場合は，#include <functional>でヘッダファイルを読み込み，

```
function<int(int)> f1 = [=](int n){ return p*n; };
```

と指定する必要があります．ラムダ式自体の戻り値の型は自動的に推定されていて，明示的に指定するのは少し面倒なので，ラムダ式の定義に名前を付ける場合はautoを使うことを勧めます．

上の説明でラムダ式の基本的な使い方はわかるはずです．しかし，なぜ，このようなものを用いるのだろうか，と疑問に思うでしょう．単なる計算であれば，計算すればよいし，関数を計算するならば，関数を定義した方がこの面倒な文法を用いるよりも簡単だろうと思うかも知れません．もっとな疑問です．ラムダ式ではスコープ内のデータを指定して使えるのが，それが文法のオーバーヘッドに見合うのか，というのはもっともな疑問です．ラムダ式に利便性が有る例を次に見てみましょう．

9.9.2 ラムダ式：応用

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main(){
  vector<double> v({0.1,1,1.1,11});
  double a(5);
  transform(v.begin(),v.end(),v.begin(),[a](double x){ return a*x; });
  cout<<count_if(v.begin(),v.end(),[](double x){ return x>1 && x<10; })<<endl;
  cout<<"v*"<<a<<": ";
  copy(v.begin(),v.end(),ostream_iterator<double>(cout," "));
```

```
}
```

上のプログラムを実行すると次の出力を得ます.

```
2
v*5: 0.5 5 5.5 55
```

transformはalgorithm内の便利な関数で，指定したイテレータ間のデータを変換します．transformの3つ目の引数は要素に働く変換の**関数オブジェクト**です．ここでは，ラムダ式を用いて，変数aをかける変換としました．count_ifは条件を満たすイテレータ間の要素数を数える関数で，3つ目の引数は条件の関数オブジェクトです．ここでは，1より大きく10より小さい，という条件にしました．作動を確認しましょう．vは要素0.1, 1, 1.1, 11を持つvectorです．transformで5倍にすると，要素は0.5, 5, 5.5, 55になり，1と10の間に2個要素があります．copy(v.begin(),v.end(),ostream_iterator<double>(cout," "));は空白を要素間に入れて，標準出力にイテレータ間の要素を表示します．これを用いて，変換されたvectorの要素を確認しています．

algorithmの関数で用いた関数オブジェクトには名前を付けて，関数オブジェクトを定義し，それを引数に使うこともできました．しかし両方とも1回しか使わない「使い捨て」なので，手間が面倒です．さらにラムダ式の方が，transform，count_if内に関数オブジェクトの中身が表示され，コードが読みやすくなります．上の例ではラムダ式の優位性は明らかでしょう．

もう1つ例を見てみましょう．

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

template <class T>
ostream &operator<<(ostream &stream,vector<T> const v){
  for( typename vector<T>::const_iterator p=v.begin() ; p!=v.end() ; p++ )
    stream << (*p) << " " ;
  return stream;
}

class Baller{
public:
  Baller(string s,int n): name(s),goal(n){}
  string name;
  int goal;
};


ostream & operator <<(ostream &stream,const Baller &b){
  stream<<"("<<b.name<<": "<<b.goal<<")";
  return stream;
}

int main(){
  vector<Baller> vb;
  vb.push_back(Baller("Miura, Kazuyoshi",55));
  vb.push_back(Baller("Kamamoto, Kunishige",75));
  vb.push_back(Baller("Hara, Hiromi",37));

  cout<<vb<<endl;
```

```
  cout<<"--------------------------------------------------"<<endl;
  sort(vb.begin(),vb.end(),[](const Baller &b1,const Baller &b2){ return b1.goal>b2.goal;} );
  cout<<vb<<endl;
}
```

サッカー選手の日本代表ゴール数をsortして表示するプログラムです．プログラムを実行すると次の出力を得ます．初めのリストと，ゴール数が多い順に並べ直したリストを表示するプログラムです．

```
(Miura, Kazuyoshi: 55)(Kamamoto, Kunishige: 75)(Hara, Hiromi: 37)
--------------------------------------------------
(Kamamoto, Kunishige: 75)(Miura, Kazuyoshi: 55)(Hara, Hiromi: 37)
```

コードではサッカー選手の名前とゴール数のクラスBallerを定義し，それをvectorに入れました．sortの3つ目の引数が比較（<）の関数オブジェクトになります．上の例では，この比較関数オブジェクトにラムダ式を用いてその場で定義しています．sortは小さい方から並べるので，ここでは逆のゴール数の多い方を「小さい」ことにして，多い順に並べました．演算子オーバーローディングの説明で，非直観的なオーバーロードはすべきでないと書きましたが，この場合は使い捨てなので，逆でも良いであろうという判断です．この場合も，単にオブジェクトのメンバデータの大小を比較するためだけに，わざわざ新たな関数オブジェクトを定義しそれを引数として使うよりも，使うのが1回だけであればその場で定義して書き込んだ方が簡単，かつわかりやすいでしょう．また，上のコードでは，vector，Ballerについて演算子<<をオーバーロードして，コードを書きやすくしています．

9.9.3 なぜラムダ式

上で見てきたように，ラムダ式を使うことには以下のような利便性が考えられます．

1. 名前を付ける必要が無い．
2. スコープ内のデータが使える．
3. algorithm等の関数で引数に関数オブジェクトを用いる場合に，わかりやすく簡潔なコードを書きやすい．

9.10 auto

9.10.1 autoとは

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

int main(){
  auto n = 0; // int
  cout<< n<<" ";
  auto m = 0L; // long int
  cout<< m<<" ";
  auto x = 0.0; // double
  cout<<x <<" ";
  auto y = 0.0L; // long double
  cout<<y <<" ";
  auto z = 0.0f; // float
  cout<<z <<" ";
  auto s = "0"; // const char*
  cout<<s <<endl;
```

```
}
```

autoキーワードはコンパイラに変数の型を判断させるキーワードです．上のプログラムを実行すると以下の出力を得ます．

```
0 0 0 0 0 0
```

どの変数もautoキーワードで処理できます．初期値よりコンパイラが推定できるからです．上のコードでは，推定されたのがどのような型であるかを各データのコメントとして付しています．表示されるのは，全て0ですが，それぞれ違う型です．型は初期値から推定しているので，たとえば

```
auto w; // コンパイルエラー
```

は型の推定のしようがないので，当然のように文法エラーです．**標準的な型で上のコードのようにautoを用いるのは勧めません．**データの型が何であるかわかりにくくなり，間違いを犯しやすくなるからです．たとえば，上のコードでsの型はstringと思うかも知れませんが，違います．この型が何であるかを確実にすぐ判定できないと，ミスをする可能性があります．また，実数についても，float，double，long doubleがあるので，どれかを判定できる必要があります．一方，double等の型を書くのには手間が大してあるわけではなく，間違いにくいコードを書く上では，データの方を認識しておく必要があります．コードを見てautoがどの型を指すのかを考える時間も無視できません．よって，上の例のような使い方は実践的なコードではわかりにくくなったり，間違いやすくなるコストに見合うだけ手間が減るメリットがありません．上の例はautoの仕組みを理解するには良い単純な例ですが，実用的ではないです．ただ，これは私の考えであって，スクリプト系言語（Python, Ruby, シェルスクリプト等）のようにデフォルトでは変数に型宣言は不要で，必要な場合に指定する言語もあるので，コーディングのスタイルや，時代に依ると思います．

autoとクラス autoは自分で定義したクラスでも用いることができます．

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

class A{
public:
  A(int n): n_(n){}
  int n_;
};

auto plusA(A a,int p){   return A(a.n_+p); }

int main(){
  A a(100);
  auto b = a;
  cout<<b.n_<<"  " <<plusA(a,10).n_<<endl;
}
```

上のプログラムを実行すると次の出力を得ます．

```
100  110
```

ここでは，関数の戻り値をautoと=で代入する場合に，でコンパイラに推定させました．ただ，A plusAやA b=a;と書くかわりにauto plusA，auto b=a;と書いているので，わかりにくくもないですが，メリットもさほど無いでしょう．

9.10.2 auto:応用

上ではautoを使うことにより，コードがわかりにくくなる点を指摘しました．しかし，autoが大変有用であり，使うことが勧められる状況がいくつか考えられので，以下で説明します．

複数の型に対応する場合

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

int main(){
  vector<double> v(10);
  generate(v.begin(),v.end(),[](){ static int j=0 ; return 0.5*j++; });
  for_each(v.begin(),v.end(),[](auto x){ cout<<x<<" "; });
}
```

上のプログラムを実行すると次の出力を得ます.

```
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5
```

for_eachを用いて，各要素を標準出力に出しています．この際，ラムダ式の引数の型にautoを使っていて，この場合だけならばdoubleとしても，あるいはした方が良いかも知れません．ただ，この場合はautoにしておけば，全く同じ指示で，<<が定義されているデータについて対応できます．なお，vector自体はラムダ式をgenerateの引数に用いて初期値を与えています．その際，staticの修飾子を使っていますが，ここではstaticを指定しないと，jが常に0になってしまい，全要素が同じで0.5にになります．なお，ここではreturnした後にjを1増やすためにj++を用いています．通常，++，--を他の指事内に含めることは勧められませんが，これは1つの例外です（5.2節参照）．

イテレータ

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

int main(){
  vector<int> v(10);
  generate(v.begin(),v.end(),[](){ static int j=0 ; return j++; });
  for(auto it = v.begin(); it != v.end(); ++it) { cout<<*it<<" "; }
  cout<<endl;
  for(auto it = find(v.begin(),v.end(),5) ; it != v.end(); ++it) { cout<<*it<<" "; }
}
```

上のプログラムを実行すると次の出力を得ます.

```
\0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
```

上のコードでは，要素が$0,1,\cdots,9$のvector，vを定義し，その中で，findを用いて5以降の要素を表示しています．表示には上ではラムダ式を使いましたが，ここでは，イテレータを用いたforループを用いています．イテレータitは型を明示するとvector<int>::iteratorになりますが，autoを使うと楽です．また，この場合もautoを使えば要素のデータ型にかかわらず，<<が定義されているデータであれば全く同じ指示で標準出力に表示できます．イテレータ型の明記は長くなることが多いので，autoを使うメリットが高く，紛らわしくもないので使うのが良いでしょう．

ラムダ式の型　前節で説明したとおり，ラムダ式は名前を付けずに使える関数です．これに名前を付けることも可能です．この場合，戻り値の型はautoを使うのが便利で，そうしないと面倒です．これは，安全性と利便性を天秤にかけたときに後者の方が勝つ場合が多いです．

Chapter 10

最後に

ここまで読んできていかがだったでしょうか？ここまでの内容を理解していれば，今後C++で実践的にコーディングしようとした時にできる基盤ができていると私は信じています．すぐに使いこなせるという意味ではありません．使いこなすためには実践による経験が必要です．ただ，C++言語のプロジェクトであれば理解はできるはずです．また，特定のプラットフォームで本格的なコーディングするには，そのAPIを読んで理解する必要があり，これが一番面倒な作業である場合が多いです．

私自身は毎日のようにプログラミングをしたり，自分で書いたプログラムを使っています．それは，研究をするために必要や有用であったり，他の作業にも便利だったりするからです．たとえば，画像ファイルが100個あって，そのファイル名の先頭に全部特定の文字列を付けたいとしましょう．皆さんだったらどうしますか？私だったら簡単なプログラムを書いて処理します．そういった，日常的な作業にもよく使います．ただ，常にC++を使うわけではありません．たとえば，先の作業だったら私はRuby，あるいはシェルスクリプトなどのスクリプト系言語を使うでしょう．

プログラミングには，その作業に適したプログラミング言語があると思います．ほぼ何でもC++で処理する，あるいは，逆にスクリプト系言語で処理する，ことも可能ですが，適材適所だと思います．私は，数値計算のようにパフォーマンスを要求される場合や，少し複雑であり間違いをしたくない場合は，C++を用います．コンパイルした言語の方が一般に処理速度が速く，また，タイプ宣言を厳しく要求される言語の方が，防衛的なプログラミング，つまり間違いを早い段階で見つけるプログラミングができるからです．また，Arduinoのような組み込みコーディングもC++で行います．逆に，一回限りの簡単な操作であれば，スクリプト系言語を使うことが多いです．その方がタイプ宣言もコンパイルも必要無く，テストし続けながら書けて手軽にコーディングができるからです．

皆さんも，余裕があれば是非とも様々なプログラミング言語を使ってみると良いと思います．便利であるばかりでなく，それにより，プログラミングの，発想法，仕方に幅が出てきてより強力な，そして楽しいプログラミングができるでしょう．私は，近年はC++，Rubyと単純なシェルスクリプトを主に用いています．これは実用性とともに個人的な好みの結果でもあるので，皆に自分の好きな言語を見つけて欲しいです．

必要でり，有用であるプログラミングですが，やはり楽しいというのが私自身にとっては大きな要素です．子どもの頃，上手ではありませんでしたが，プラモデルやラジオなどを作るのが好きでした．プログラミングには，それに通じるものづくりの楽しさがあるといつも感じます．自分の書いたものが目の前で動くのには達成感と感動があります．皆さんにも，是非ともプログラミングを楽しんで欲しいと思います．

Chapter 11

プログラミングやC++についてさらに学びたい方へ

以下に，さらにプログラミング，C++について学びたい方への参考文献とURLをいくつかあげます．あくまでも私が主観的に選択したいくつかの例です.

“Effective C++”, Scott Meyers (Addison-Wesley Professional, 2005)，小林健一郎訳 (丸善出版, 2014)　C++言語の基本を理解している人に，C++を使うにあたってのコツ，間違いをしにくくする方法などを解説しています．実践的な本で，コーディングをするにあたって示唆に富んでいます.

“Numerical Recipes 3rd Edition: The Art of Scientific Computing”, William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery (Cambridge University Press, 2007).「C言語による数値計算のレシピ」，丹慶勝市他訳　方程式の解法，微分，積分，最小化問題，等の様々な数値計算の問題の解法について説明しています．アルゴリズムとその論理の説明が明快で，数値計算に関するバイブルと言えるでしょう．コード例もありますが，そのまま使う必要はなく，参考にして自分に合ったスタイルで書き直すと良いでしょう．原書はC++言語用の本ですが，C言語の版の翻訳書しかないようです.

“C: A Reference Manual”, Samuel P. Harbison, Guy L. Steele Jr (Pearson, 2002);「S・P・ハービソン3世とG・L・スティール・ジュニアのCリファレンスマニュアル」，玉井浩訳（エスアイビーアクセス，2015）　実際にコーディングする際には，`double`型の精度は何桁あるのか，`int`型はどの範囲の整数を格納できるのか，といった情報は重要です．一般にはこれらはコンパイラ，プラットフォームの実装に依存しますが，最低限のANSI基準があり，「まとも」なコンパイラはこれに準拠しています．そのANSI基準を説明している本です．C言語に関する本ですが，内容はC++言語にもあてはまります．気になった時にすぐ調べられるようにプログラミングする際にそばにあると便利な本です.

“The C++ Programming Language”, Bjarne Stroustrup (Addison-Wesley Professional, 2013);「プログラミング言語C++」柴田望洋訳（SBクリエイティブ, 2015）　C++を設計した本人著の本です．厚い本で，初めから最後まで読むには適しているとは私は思いませんが，C++に関してわからない事があった際に調べると大いに参考になることがあるでしょう．C++についての悩みを解決してくれる本です.

「ゼロからよくわかる! Arduinoで電子工作入門ガイド」，登尾徳誠（技術評論社，2018年）　C++を使えば容易に，そして安価に組み込みプログラミングを楽しめます．Arduinoはロボコン等でも定番のマイコンボードです．プログラムをC++で書いてボードにアップロードすれば，稼働します．単に目の前のコンピューターでプログラムを走らせるだけではなく，独立した機械でプログラムを走らせることができるのが組込プログラミングです． Arduinoの素晴らしいところは，C++でコーディングできて，IDEが無料で使いやすく，組み込みプログラミングが非常に容易にできることです．C++言語を使えれば，1日あれば色々な事ができるはずです．たとえば，LEDを様々なタイミングで光らせる，音を出す，温度センサー

等様々なセンサーを使う，そしてその情報を元にLEDを光らす，モーターを動かす等がすぐにできます．上にあげた本は一例で，Arduinoの導入の仕方を説明した本があればすぐに使いこなせるはずです．

Boost （https://www.boost.org/）
様々な分野の質の高いC++のライブラリの集まりです．擬似乱数，特殊関数，有理数計算，線形代数，等の数値計算やメタプログラミング等の抽象的な分野も網羅しています．無料で使えます．

GNU Scientific Library (GSL) （http://www.gnu.org/software/gsl/）
数値積分，微分，線形代数等の数値計算のライブラリです．無料で使えますが，GSLを含んだソフトウェアを公開する場合は，公開する義務があるので，ライセンス（GPL）を確認して下さい．

"C Programming Language", Brian W. Kernighan, Dennis Ritchie （Prentice Hall，1988）；「プログラミング言語C」石田晴久訳（共立出版，2018） C言語，そしてUnixをデザインして造った人達によるC言語の解説です．C++言語以前の本で，C++には全く言及していません．しかし，C言語はC++言語の基盤であり，その哲学，そして，プログラミングの仕方に関しては今でも参考になります．厚くない本ですが，密度は濃いです．

"The Elements of Programming Style", Brian W. Kernighan, P. J. Plauger （McGraw-Hill, 1978） 意図的に短い本です．題名どおり，プログラミングをする際の作法を解説しています．文法的には正しくても，勧められない書き方はあり，理由を含めて，１つのプログラミング言語に限定せずに，説明しています．「それはそうだよね」感が強くあった事が記憶に残っています．プログラミング言語に関わらず参考になります．

"The Art of Computer Programming", Donald E. Knuth (Addison-Wesley Professional, 2011); 有澤誠他訳（KADOKAWA，2017） アルゴリズムに関する古典的名著です．学術的で，内容量も非常に多いので，専門家になりたい人以外は全部を読むことはないでしょう．一般的なアルゴリズムついて調べるには論理的で洞察に富む本です．論理的かつ厳密ですが，少し敷居が高い本かも知れません．

Qt （https://www.qt.io/）
QtはWindows，MacOS，LinuxのどのOS上でも機能するプラットフォームです．GUIを含む豊富なライブラリが備わっています．同じコードで複数のOS上で走るアプリケーションを造れます．Android，iOSのスマートフォンやタブレット上でも走るように開発できます．統合開発環境（付録A参照）もあり，無料で使えるオープンソース版も存在します．

Appendix A

コンパイル，プログラム実行，とIDE

A.0.1　コンソールでの環境

第1章で説明したように，コードを書いてから，コンパイル，リンクし，実行します．具体的な手順とツール類を簡単に紹介します．まず，最低限のCUIであるコンソールでの環境の例を説明しますが，次に説明するIDEを用いた方が楽に実行できるでしょう．

コードはテキストファイルなので，どのエディタでも編集できます．ただ，編集する際には，C++の文法を認識するエディタを使った方がはるかに楽です．文法（コンテクストともいいます）を認識すると，色分けをしたり，読みやすいように体裁を整えたりするので編集しやすくなります．IDE付属のエディタも文法を認識しています．

Windows環境　コンテクストを認識するエディタには，IDE付属のエディタ以外にTeraPad，Emacs等があります．コンパイル，リンクはMinGW（Minimalist GNU for Windows）で行えます．コンソールには Windows のPowerShell，コマンドプロンプトが使えます．いずれも標準か無料です．

MacOS環境　コンテクストを認識するエディタには,IDE付属のエディタ以外にAtom，Bluefish，Emacs等があります．コンパイル，リンクはXCodeのパッケージをインストールすれば行えます．コンソールには標準のTerminalが使えます．いずれも標準か無料です．

A.0.2　IDE（統合開発環境）

IDE（Integrated Development Environment）は統合開発環境 ともよばれ，編集，コンパイル，リンク，プログラム実行まで全てこの開発環境の中で行えます．よって，楽です．特に初心者はIDEを使うと良いでしょう．

Qt Creator　IDEには様々なものがありますが，Qt CreatorはWindows，MacOS，Linux環境の全てで使えるIDEなので，例としてあげます．Qtは同じコードでGUIアプリケーションを含め，Windows, MacOS，AndroidやiOS端末（iPhone, iPad等）用のアプリケーションを造れるフレームワークです．フレームワークは様々な機能を持つソフトウェアライブラリの集合です． Qt Creator はQtフレームワークのIDEです．勉強用途やオープンソースのコードを書くためであればQtは無料です．Qtで作ったものを公開する場合は，ライセンスの条件について確認して下さい．

`http://www.qt.io/download-open-source/`

より無料のオープンソース版をダウンロードできます．

MacOSにQtを導入する場合は，まず，XCodeを導入してからQtをインストールします． XCodeはMacOS専用の純正IDEで，無料です．MacOSでMacOS用だけのコードを書く場合にはXCodeが適しているでしょう．細かいことですが，Qtを使う場合には，プロジェクトのフォルダ名，ファイル名は全て1バイト文字にしてください．そうしないと，コンパイラがプロジェクトを見つけられない場合があります．

Index

#define, 103
#ifdef, 104
#include, 102
&&, 37
(:?), 74
*=, 42
++, 42
+=, 42
--, 42
-=, 42
->, 55
/=, 42
<, 37
<=, 37
==, 37
>, 37
>=, 37
const_iterator, 94
false, 36
complex, 92
1バイト文字, 5
2バイト文字, 5

int64_t, 9

algorithm, 94
auto, 111

bool, 9
boost, 49

char, 9
Character User Interface, ↪ CUI
class, 25
close(), 72
const, 31
c_str(), 71
CUI, 4, 118
C文字列, 71

delete, 55
do while, 45
double, 8

else, 39
else if, 39

explicit, 102

false, 39
final, 90
float, 9
for, 44
friend, 26

Graphical User Interface, ↪ GUJ
GUI, 4

IDE, 118
if, 36
ifstream, 69
includeガード, 105
int, 8
int_least8_t, 9
iterator, 93

long, 9

namespace, 21
new, 55

OS, 1
override, 79

pow, 17
private, 26
public, 26
public継承, 78
pure virtual関数, 86
push_back, 62

Qt, 117

short, 9
signed, 9
size_type, 60
STL, 93
string, 8
struct, 34
switch, 46

\t, 41
true, 36, 39
typename, 95

typename, 92

uint64_t, 9
uint_least8_t, 9
unsigned, 9

vector, 59
virtual関数, 79
void, 16
void, 9

wchar_t, 9
while, 42

XCode, 118

値
　変数の---, 7
値渡し, 64
入れ子, 41
演算子, 11, 36
　---の優先順位, 38, 39
　---のオーバーロード, 96
親クラス, ↪ 基底クラス
科学表記, 107
拡張子, 4
仮想関数, ↪ virtual関数
型, 7
関数
　---のオーバーロード, 19
関数オブジェクト, 105
関数ブロック, 99
機械語, 1
基底クラス, 76
基本クラス, ↪ 基底クラス
偽, 36
擬似乱数, 48
　の種, 48
繰り返し, 42
継承, 26, 76
構造体, 34
高レベル, 1
再帰的, 16
最適化, 67
参照, 53
参照渡し, 64
算術演算子, 11
修飾語, 9
出力ファイルストリーム, 68
真, 36
実体, 25
実体化, 25
自動変換, 101
条件判定, 36
数値積分, 107
宣言, 7, 27
全角文字, 5
添字, 51
多重継承, 78
多相性, ↪ ポリモーフィズム
多態性, 77, 82, ↪ ポリモーフィズム
代入, 7
抽象クラス, 86
定義, 27
統合開発環境, 118
動的メモリ確保, 52
名前空間, 21
入力ファイルストリーム, 69
配列, 51, 60
派生クラス, 76
半角文字, 5
汎関数, 107
反復子, 93
番地, 53
比較演算子, 37
引数, 15
非インライン定義, 27
複合関数, 16
複素数, 92
変数, 7
　---の値, 7
変数名, 14
防衛的プログラミング, 32, 45
無名関数, 109
文字列, 4
戻り値, 15
優先順位↪ 演算子の優先順位 38
要素, 51
乱数, 48
　の種, 48
累乗, 17
論理演算子, 37
アクセス制御, 26
アドレス, 53
アプリケーション, 1
アルゴリズム, 94
イテレータ, 93
インスタンス, 25
インデックス, 51, 60
インライン定義, 27
エンコーディング, 6
オブジェクト, 25
オペレーティングシステム, 1
オーバーライド, 79, 81
オーバーロード, 19, 79
　演算子の---↪ 演算子のオーバーロード 96

キャプチャ，109
クラス，25
クラッシュ，11
グローバル変数，99
グローバルデータ，99
ゲッター，29
コメント，5
コメントアウト，11
コンストラクタ，28，80
コンソール，4
コンテナ，93
コンパイラ，1
コンパイル，1
コーディング，3
コード，1，3
シンプソン公式，108
スコープ，21，33，45，99
スーパークラス，↪ 基底クラス
セッター，29
ソフトウェア，1
ソースコード，↪ コード
ソース・レベルデバッガ，24
タブ，41
ターミナル，↪ コンソール
ダメ文字，6
テンプレート，60，91
ディレクトリ，68
デストラクタ，80
デバッガ，24
デバッギング，24
デバッグ，11
デファレンス，53
デフォルト引数，18
デフォルトコンストラクタ，29
ネスティング，41
ハードウェア，1
バイト，5
バグ，11，24
ビット，5
ビット演算，12
ファイルストリーム，68
ファンクタ，↪ 関数オブジェクト
フォルダ，68
フレームワーク，118
ブロック，21，36，99
　関数 ---↪ 関数ブロック 99
ブール代数，39
プリプロセッサ，102
プログラミング，1，3
プログラム，1
プログラムエラー，11
プロファイラ，67
ヘッダファイル，20
ポインタ，53
ポインタ演算，58
ポリモーフィズム，77，82
マイクロコード，↪ 機械語
マクロ関数，103
マルチバイト文字，5
メソッド，↪ メンバ関数
メッセージ，↪ メンバ関数
メッセージング，26
メモリーリーク，33
メンバ関数，25
メンバデータ，25
ラムダ式，108
リンク，1
ループ，42
ループカウンタ，43，44，52
レファレンス，53
レベル，3