技術職員専門課程研修（平成22年度）
データ処理のためのC++入門

高エネルギー加速器研究機構
藤井啓文

目 次

第1章　はじめに

本書は高エネルギー加速器研究機構の平成 22 年度第一回技術職員専門課程研修で行なわれたコンピュータプログラミング言語 C++ の入門の講義録です。講義では説明不足だった分の増補や順番の若干の入れ替えが行なわれています。本講義の表題は単に「C++ 入門」とせずに、「データ処理のための」という枕詞をつけました。プログラミング言語というのは、利用者にとっては、あくまで道具です。道具は何らかの目的のために使うのであり、ここで想定する読者は「データを処理する」ためにコンピュータを使う利用者です。

C++ は多種多様な目的に対し汎用に使える言語として設計されています。その規格は膨大であり、標準として用意されているライブラリも豊富にあります。利用者にとって大事なことは、これらを全部知ることではなく、問題解決のために必要とする機能やライブラリをいかに利用するのかということです。そのためには「習うより慣れろ」が極めて有効であることは言うまでもありません。しかし、初心者には更に「慣れる」ためのきっかけが必要です。いきなり参照マニュアルを読んでも何のことかさっぱりわからないでしょう。一方、入門書の多くは、規格を噛み砕いて、更に実例を示しながら解説しています。これはこれで役に立つのですが、時間がかかります。

本書は、データ処理に利用することを目標にして、できるだけ手短かに C++ 言語を解説していきます。

1.1　準備するもの

本書を読むにあたって、ここに例示してあるプログラムを実際に自分で書いてみて実行しながら理解していくことを強く勧めます。道具は使うことではじめて価値が出るものです。

実際に自分で書いて実行してみるためには、以下のものを準備する必要があります。

- C++コンパイラ
- テキストエディタ

このうち、テキストエディタに関しては、普通のテキストファイルが編集できるものであれば何でも構いません。本書で示すのは小さいプログラムですので、例えば Windows の場合であれば「メモ帳」でも十分です。

コンパイラも ISO/JIS 規格準拠のものであれば何でもよいのですが、ここでは代表例として GCC (GNU Compiler Collection) の g++ を使っていきます。Linux OS であれば GCC は大抵インストールされています。Windows ユーザの場合は、`cygwin` または `MSYS/Mingw` をインストールすると GCC も一緒にインストールすることができます（初めての人には後者を薦めます)。

GCC が正しくインストールできているかどうか使う g++ のバージョンを調べましょう。g++ に `--version` をつけて

```
g++ --version
```

のように起動します。例として Windows の MSYS/Mingw の場合を応答を示します。

```
g++ (GCC) 3.4.5 (mingw-vista special r3)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

1.1.1 Windows の場合の他の選択肢

Windows の場合は、Microsoft も無償の開発環境 Microsoft Visual C++ Express Edition を配布していますので、GCC の代わりに、これを使うのもよいでしょう。筆者は Express Edition 2008 を使った経験がありますが、本書を執筆している時点では Express Edition 2010 も利用できるようになっているようです。

1.2 統合開発環境

プログラムが大規模になってくると、扱うファイルの数も、コードの量も増え管理すべきものが多くなって大変です。このような管理機能も統合した開発環境も無償で入手可能です。上に述べた Microsoft Visual C++ Express Edition も統合開発環境です。Linux でも動く無償の統合開発環境としては Eclipse が有名です。

第2章　とりあえず動かしてみよう

まずは一連の手続きを理解するために、簡単なプログラムを書いて実行してみましょう。

2.1　まずは hello world

いかなるコンピュータプログラミング言語の学習であれ最初に書くべきプログラムは表示画面に hello world と表示するプログラムに決まっています。

これは C 言語の創始者である Brian W. Kernighan と Dennis M. Ritchie による有名な教科書、The C programming language [1] に出てくる最初の program で、これ以降、世の中の様々なコンピュータプログラミング言語の最初の例題として扱われています。C で書くとプログラム 2.1 のようなプログラムです。この C のプログラムは、このままでも正しく実行される C++ のプログラムでもあるのですが、せっかくですので、ここではもう少し C++ らしいプログラム 2.2 に示すプログラムを書くことにしましょう。プログラムの説明は後回しにして、実行するところまで概観してみます。

2.1.1　プログラムをテキストエディタで書く

まずは、適当なテキストエディタを使って上記のプログラムを書いたファイルを作ります。テキストエディタは何でもよいのですが、作られるファイルはただの単純なテキストファイルである必要があります。文字飾りとかフォント指定とかの無い単純素朴なテキストファイルでなければなりません。

ファイルの拡張子

ファイル名の拡張子ですが、C の場合は通常は `.c` です。規格として決まっているわけではありませんが、これはほぼ定着しています。

では C++ の場合はどうかというと、いくつかの拡張子が使われているのが現状です。ここでは `.cpp` としましょう。これは Microsoft の C++ の既定の拡張子がこうなっているので、こうし

プログラム 2.1: C の hello world

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("hello, world\n");
5     return 0;
6 }
```

プログラム 2.2: C++ の hello world

```
1 #include <iostream>
2 int main()
3 {
4         std::cout << "hello, world" << std::endl;
5         return 0;
6 }
```

ておくのが無難であろうという程度の意味です。気に入らなければ他の拡張子を使っても構いません。他によく使われる拡張子としては、.C 、 .cc や .cxx などがあります。ただし、.C はファイル名の大文字と小文字を区別しないシステム（例えば Windows）では C の拡張子である .c と区別されないので様々なシステムへの配布を考慮する場合には使わない方がよいでしょう。

ソースファイル

さて、では上記プログラムテキストエディタで書いて hello.cpp というファイルに保存しましょう。このとき、一番最後の行、6 行目の } の後に改行を入れるのを忘れないでください。このように文字で記述されたプログラムをソースプログラム（source program）、その記述したテキストファイルをソースファイル（source file）と呼びます。ここでは hello.cpp がソースファイルです。

2.1.2 実行形式のファイルを作る

次にやるべきは、このソースファイルから実行形式のファイルを作ることです。以下のようにタイプします。

```
g++ hello.cpp
```

すると、Linux であればカレントディレクトリに a.out というファイルが、また Windows の場合はカレントフォルダに a.exe というファイルが作られます。これが実行形式のファイルです。

2.1.3 実行する

実行形式のファイルができたので、実行は簡単です。Linux の場合は

```
./a.out
```

とタイプします。Windows の場合は

```
.\a.exe
```

とタイプ（.exe は省略してもよい）すれば、実行されて表示端末に

```
hello, world
```

と表示されます。

2.2 プログラムの概要

さてここで、C++ のプログラムがどのようなものであるのか、今の例題を使って簡単に説明しましょう。

2.2.1 ヘッダの読み込み

まず 1 行目にある

```
#include <iostream>
```

ですが、これは iostream という名前のヘッダを読み込めという指示です。ヘッダというのは、プログラム中で使われる関数や変数などの属性を記述したもので通常はファイルです。<> で囲まれているものは、システムとして用意してあるものです。つまり今の場合、「システムとして用意してある iostream というヘッダを読み込め」となります。この <iostream>には、入出力関連の関数や変数などの属性が記述されています。

2.2.2 main 関数

次の行は

```
int main()
```

です。最初にある int は整数型であることを示すもので、C++ の予約語の一つです。これに続く main は単なる名前ですが、その後に () が続いているので、関数の名前になります。

関数は、一般に引数を持っていて、値を受け取り、処理して結果の値を返します。関数名の後の () の中にその引数を書きますが、今の場合何も書いてないので、引数はありません。引数が無くても、関数であることを示すために () をつけます。最初にある int は、この関数は整数を返す関数であることを意味します。

この行に続く { から最後の行の } までを、この関数の本体（body）と言います。今の場合は、関数 main の本体です。

C++ の規格ではプログラムを実行すると必要な初期化を行った後、最初にこの main 関数を呼び出すことになっています。つまり、main 関数は絶対に用意しなければならない関数です。

2.2.3 文

関数の本体には、この関数がどのような機能を果たすのかを記述します。ここでは

```
    std::cout << "hello, world" << std::endl;
    return 0;
```

が、この関数 main が果たす機能です。中にセミコロン ; が 2 箇所ありますが、これは文（statement）の区切りを示します。文（statement）は、機能を実行させる一つの単位です。ここに示すように、プログラムは文の集まりとして記述されます。ここに示した main 関数の本体は二つの文から構成されています。

オブジェクト

C++ では演算や操作の対象となる実体をオブジェクト（object）と呼びます。数式で言えば項に相当するものです。

関数本体の最初の文

```
std::cout << "hello, world" << std::endl;
```

には以下の 3 つのオブジェクトがあります。

- `std::cout`
- `"hello, world"`
- `std::endl`

最初の `std::cout` は、標準出力オブジェクトを表す名前です。標準出力オブジェクトとは大雑把に言えば、プログラムを実行させている表示端末を表す名前です。名前の規則は後で述べますが、間に記号 `::` があって若干違和感を感じるかも知れませんが、これ全体で標準出力オブジェクトの名前です。

次の `"hello, world"` は、文字列オブジェクトと呼ばれるものです。二重引用符 `"` で囲むことで二重引用符の内部（二重引用符は含みません）が文字列オブジェクトであることを表しています。

最後の `std::endl` というオブジェクトですが、これは操作子（manipulator）と呼ばれるもののうちの一つです。操作子は入出力の形式を整えるためのもので `std::endl` は改行を行います。操作子にはいろいろなものがありますが、それについては後で述べます。

2.2.4 演算子

さて、

```
std::cout << "hello, world" << std::endl;
```

の中にある `<<` 記号ですが、これは演算子（operator）と呼ばれるものです。もうすこし詳しく言うと二項演算子（binary operator）と呼ばれるもので足し算 `+` や引き算 `-` の仲間です。二つの `<` を続けて書くことで一つの演算記号です。

二項演算子は、その右側と左側に演算の対象となるもの（項）がきます。二つの整数 1 と 2 の足し算を書くのに足し算記号 + を真中にはさんで $1+2$ と書くのと同じです。上の文には、この二項演算子`<<`が二つありますが、これも三つの整数 $1, 2, 3$ の足し算を $1+2+3$ と書くのと同じです。

この `<<` 演算子の演算の順序は足し算と同様左から右です。つまり、まず

```
std::cout << "hello, world"
```

が実行され、その結果に対し

```
<< std::endl
```

が実行されます。

2.2.5 式

オブジェクトやオブジェクトに対する一連の操作を記述したものを式（expression）と呼びます。通常、式は結果の値を持ちます（値を持たない式も存在します)。この結果の値を得ることを式を評価すると言います。オブジェクトそれ自身でも式です。この時の値は、そのオブジェクトの値です。

```
std::cout << "hello, world"
```

は二つのオブジェクト（std::cout と "hello, world"）と一つの演算子（<<）から成る式です。また

```
std::cout << "hello, world" << std::endl
```

も式です。

さて、この演算子<<ですが、標準出力オブジェクトが左側にある時には、右側のオブジェクトを整形して出力します。また、その結果は標準出力そのもの（正しくは標準出力オブジェクトへの参照ですが、ここでは標準出力オブジェクトそのものと理解して構いません）です。

つまり、二番目の << の部分、

```
<< std::endl
```

の左側のオブジェクトは（"hello, world" を出力した後の）標準出力オブジェクトです。つまり

```
std::cout << "hello, world";
std::cout << std::endl;
```

と二つの文に分けて書いても、同じ結果になります。

2.2.6 関数から戻る

main 関数の本体の二番目の文に移りましょう。二番目の文は

```
return 0;
```

です。ここで return というのは、C++の予約語です。予約語というのは予め機能や使い方が決められている単語です。この return は関数から関数を呼び出した側へ制御を戻すものです。その際、後ろにこの関数が返すオブジェクトをを書きます（何も返さない関数というもの存在し、その場合には何も書きません)。main 関数 の場合は、整数を返さなければならないので、ここでは値 0 の整数オブジェクトを返してます。

規格では return の後ろに書くのは式となっています。今の場合、（整数）オブジェクトそのものを書いてますが、前節で説明したように、オブジェクトはそれ自身で式ですので、これで規格を守っていることになります。

さて、main 関数を呼び出した側に制御を戻すとはどういうことでしょう？一体誰が main 関数を呼び出したのでしょう？

これは場合により様々です。コンパイラを使って実行形式のファイルを作り、それをキーボードを叩いて実行させたかも知れませんし、あるいはバッチファイルを作ってその中から実行させたかも知れません。場合によっては 2.6 で述べる C++インタープリタを使って実行させたかも知れません。いずれにせよ、このプログラムを実行させた環境があるわけで、これを実行環境と呼びます。main 関数から戻ると、適当な後処理の後、実行環境に制御が戻ります。この時、main 関数

が戻す整数値が実行環境に渡されますが、それがどのように使われるかは実行環境に依存します。バッチ処理であれば、その戻った値により更に処理を継続するとか、特定の処理をスキップするとかの記述ができることもあるでしょう。用途は特定しないけれども自動継続処理や分岐処理ができるようになっているわけです。

このように main 関数の戻す値を、どう使うかは実行環境に依存しますが、値 0 または <cstdlib> にある EXIT_SUCCESS が戻された場合は正常終了を、<cstdlib> にある EXIT_FAILURE が戻された場合は異常終了を示すことになっています。

プログラム 2.3: 整数の表示

```
1 #include <iostream>
2 int main()
3 {
4         std::cout << 123 << std::endl;
5         return 0;
6 }
```

プログラム 2.4: 8 進表記で与えた整数の表示

```
1 #include <iostream>
2 int main()
3 {
4         std::cout << 0123 << std::endl;
5         return 0;
6 }
```

2.3 数値を扱う

さて、前節では文字列を標準出力に表示しました。ここでは数値の表示を行ってみます。

2.3.1 整数の記法

まずは簡単な整数の表示から行いましょう。プログラム 2.3 を見てください。前のプログラム 2.2 との違いは文字列オブジェクト `"hello, world"` のところに `123` と書いてあるだけです。このように `0` 以外の数字から始まって一連の数字が並んでいるオブジェクトは整数オブジェクトで、その値は 10 進数での値として扱われます。ですから、このプログラムの最初の二項演算は標準出力オブジェクトに 10 進数で 123 の値を持つ整数オブジェクトを出力せよという意味になります。何をややこしい言い方をしているのだ、書いた通りに出力されるだけじゃないかと思うかも知れません。

ではプログラム 2.4 の例はどうでしょう？結果は `83` と表示されます。C++ の整数の記法で `0` から始まり、`0-7` の数字が続くオブジェクトも整数オブジェクトとして扱われますが、その値は 8 進表記されているものとして扱われます。整数を演算子 `<<` を使って標準出力に出力した場合、（他に何もしなければ）10 進表記で表示されます。ですから今の場合、ソースファイルに書かれた 8 進表記の整数が 10 進表記で表示されたために `83` となったわけです。このプログラム上で生成されているのは、一つの整数オブジェクトであって文字列ではありません。

ついでに 16 進表記も述べておきましょう。プログラム 2.5 は 16 進表記で整数オブジェクトを与えたものです。実行結果は、前と同様に 10 進表記で `291` となります。このように先頭 2 文字が `0x` または `0X` で始まり、その後に 16 進数を表す文字（つまり `0-9`, `A-F`, `a-f`）が連なった表記は 16 進表記による整数オブジェクトを表します。

さて、ここまでは単一の整数オブジェクトを与えていましたが、結果として整数オブジェクトが与えられる場合について考えてみます。二つの整数の足し算した結果は一つの整数オブジェクトを与えます。プログラム 2.6 を見てください。二つの整数オブジェクト `2` と `0x0f` を二項演算子 `+` で結合しています。二項演算子 `+` は二項演算子 `<<` よりも優先度が高いので先に実行されます（演算子の優先順位については付録 D を見てください）。つまり、先に `+` が行われ、その次に `<<` が行わ

プログラム 2.5: 16 進表記で与えた整数の表示

```
1 #include <iostream>
2 int main()
3 {
4         std::cout << 0x123 << std::endl;
5         return 0;
6 }
```

プログラム 2.6: 演算結果の整数の表示

```
1 #include <iostream>
2 int main()
3 {
4         std::cout << 2 + 0x0f << std::endl;
5         return 0;
6 }
```

れます。もちろん（2 + 0x0f）のように括弧 () を使って優先順位を明示しても構いません。このプログラムの実行結果は（10 進表記の 2 と 16 進表記の 0x0f すなわち 10 進表記で 15 が足されて）17 と表示されます。

2.3.2 浮動小数点数の記法

数値計算を行なう場合、実数を扱いたい場合が数多くあります。実数は連続なので多くの場合、有限のビット数では正確には表すことができないのですが、近似値でよい場合には C や C++ では浮動小数点数を使うことができます。

浮動小数点数は次のように記されます。

- 整数部
- 小数点（.）
- 小数部
- 指数部
  - 文字 e または E
  - 符号+ または -（省略可、省略時は +）
  - 整数の指数

指数部は 10 のべき乗を表します。このうち、整数部または小数部のどちらか一方は省略可能です。また指数部を省略することはできますが、その場合は小数点は省略できません。整数部のみで小数部が無い場合、指数部があれば、小数点と小数部は省略できます。プログラム 2.7 は 3 通りの浮動小数点数記法で 2.71828 を表している例です。

プログラム 2.7: 浮動小数点数の表示

```
1 #include <iostream>
2 int main()
3 {
4         std::cout
5                 << 271828e-5 << std::endl
6                 << 2.71828 << std::endl
7                 << 0.0271828E2 << std::endl;
8         return 0;
9 }
```

プログラム 2.8: C++ による華氏-摂氏温度表

```
1  /* fahren.cpp
2     prints fahrenheight vs celcisus table */
3
4  #include <iostream>
5  int main()
6  {
7          int lower(0), upper(300);        // Lower and upper limits
8          int step(20);    // Step
9
10         int fahr = lower;
11         while (fahr <= upper)
12         {
13                 int celsius = 5 * (fahr - 32) / 9;
14                 std::cout << fahr << '\t' << celsius << std::endl;
15                 fahr = fahr + step;
16         }
17         return 0;
18 }
```

2.4 変数、算術演算と繰り返し

さて、もう少しプログラムらしい例題として、ここでも K&R の C の教科書 [1] に習って、華氏の温度と摂氏の温度の対応表を作ってみましょう。プログラム 2.8 は華氏と摂氏の温度表を印字するプログラムです。これを実行すると、

```
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

という結果が得られます。

2.4.1 コメント

このプログラムの先頭2行（1行目と2行目）はCスタイルのコメントです。Cスタイルのコメントは /* ではじまり、*/ で終わります。この間に書かれた文字は翻訳時にはすべて無視されます。プログラムの実行には全く関係ありません。プログラムの説明などを書いておきます。

7行目と8行目ではC++スタイルのコメントを用いてます。C++スタイルのコメントは // ではじまり、改行で終わります。こちらも実行には全く影響を与えません。

2.4.2 宣言と定義

プログラムの7行目と8行目では、3つの変数 uppor、lower、step を導入しています。変数というのは、値を格納しておくことができる記憶域です。各々の記憶域を区別するのに、このように名前（変数名）で区別します。

C++ では扱うデータが整数であるのか小数点数であるのか、あるいは文字列であるのかなどにより扱いが区別されます。7行目と8行目の先頭にある int は、これに続く変数は整数のデータを扱うものであることを述べています。このようにデータの持つ共通の性質を型（type）と呼び、変数がどのような型のデータを扱うものであるのかを表すことを宣言（declaration）と呼びます。7行目では、int の後に二つの変数 upper と lower をカンマで区切って書いてます。このように同じ型の変数はカンマで区切って並べて宣言することができます。

宣言のうち、記憶域の確保を伴うものを定義（definition）と呼びます。ここでの7行目と8行目は変数のための記憶域をここで確保していますので、定義です。

2.4.3 変数の初期化

変数の初期化は7行目、8行目にあるように、変数名の後に値や式を () でくくることでできます。これを関数形式の初期化と言います。一方、10行目や13行目にあるように = の右側に設定する値や式を書くことでもできます。これを代入形式の初期化と呼びます。この10行目や13行目と、後で説明する（15行目で使われている）代入文はよく似ていますが、代入と初期化では意味が異なることに注意します。

2.4.4 繰り返し

11行目の while は繰り返しを指示するもので、

```
while（条件） 文
```

という構文をとります。”条件” の部分は bool 型の値、つまり true か false の二つの値のどちらかをとる式として評価され、これが true の間、”文” が実行されます。”文” は複数の”文”を{}で囲むことで、一つの”文”（複文と呼びます）とすることができます。ここでは”文”の部分は、12行目から16行目までの {} で囲まれた部分になります。

11行目の”条件”部分は

```
fahr <= upper
```

表 2.1: 数値の比較を行う二項演算子

演算子	true になる条件
>	左オペランドの値が右オペランドの値より大きい
>=	左オペランドの値が右オペランドの値より大きいか等しい
<=	左オペランドの値が右オペランドの値より小さいか等しい
<	左オペランドの値が右オペランドの値より小さい
==	左オペランドの値と右オペランドの値が等しい
!=	左オペランドの値と右オペランドの値が等しくない

表 2.2: 算術演算に使われる二項演算子

二項演算子	演算
*	乗算
/	除算
%	剰余算
+	加算
-	減算

です。これは二項演算子 <= と、その左オペランドに整数型変数 fahr、右オペランドに整数型変数 upper を持つ式です。この演算子は、右オペランドと左オペランドの数値を比較して、左オペランドの値が右オペランドの値より小さいか等しい場合には true の値を、そうでなければ false の値を持ちます。

数値として比較を行う二項演算子を表 2.1 に示しておきましょう。なお、この中で == は等値演算子と呼ばれ、他の 4 つより優先度は低くなっています。

2.4.5 算術式

13 行目は 2.4.3 で述べた代入形式の初期化です。この最後にある式は算術式で、通常の算術演算の記法とほぼ同じように書くことができます。算術演算に使われる二項演算子を表 2.2 に示しておきます。通常の算術演算と同様、乗算と除算の優先度は加算と減算の優先度より高くなっています。また () で囲むことで優先度を変更することができるのも、通常の記法と同じです。

ここで、整数同志の除算では整数未満は切り捨てられることに注意します。13 行目の等号の右辺の式は整数の演算式であるために、9 での除算が最後に行われるようにしています。こうしないと途中で切り捨てが起こり正しい値が得られません。

2.4.6 スコープ

13 行目で定義されている変数 celisius の有効範囲は 16 行目までです。つまり while の本文の中だけです。このように着目している識別子の有効範囲をスコープ（scope）と呼びます。変数名のスコープはその変数の定義が含まれるブロックとその内側です。もし内側のブロックで更に同じ名前が定義されると、外側の名前は隠されます。今の場合、while 文の外側に int celsius = 0;

とあっても `while` 文の本体には何ら影響ありませんし、また `while` 文から抜けた時点で、内側の `celsius` の値が外側の `celsius` の値に影響を与えることもありません。

プログラム 2.9: 標準入力からの読み込み

```
1  #include <iostream>
2  #include <string>
3  int main()
4  {
5          std::cout << "Please␣enter␣your␣name" << std::endl;
6          std::string yourname;
7          std::cin >> yourname;
8          std::cout << "Hello␣" << yourname << "!" << std::endl;
9          return 0;
10 }
```

2.5 標準入力からの読み込み

さて、ここまでの例題は出力するだけのものでした。ここでは入力の例題をやってみましょう。標準出力オブジェクトがあるなら、当然、標準入力オブジェクトもあります。標準入力オブジェクトも標準出力オブジェクト同様ヘッダ <iostream> で定義されていて、std::cin で表します。通常は、プログラムを起動した端末のキーボードが標準入力オブジェクトとして割り当てられています。

2.5.1 標準入力からの文字列読み込み

プログラム 2.9 は標準入力から文字列を読み込む例です。このプログラムを実行すると、

```
Please enter your name
```

と表示して入力待ちになります。キーボードから

```
Fujii
```

と打ち込むと

```
Hello Fujii!
```

と表示します。

文字列のためのヘッダ

プログラム 2.9 の 2 行目には、

```
#include <string>
```

が新たに加わってます。これも <> で囲まれてますので、システムが用意しているヘッダです。文字列関連の情報が書かれたヘッダです。ここでは std::string という型（クラス）を導入するために、これが必要です。

変数の導入

計算機で処理するデータはすべて記憶装置（メモリ）に置く必要があります。最初の プログラム 2.2 でも文字列 hello, world は、まず計算機のメモリ上に置かれ、そのメモリのデータが出力装置へ転送され表示されることで人間が読めるようになるわけです。

入力を扱うには、そのデータを計算機上で記憶しておく場所を前もって確保する必要があります。6 行目の

```
    std::string yourname;
```

が記憶する場所を確保しています。ここでは std::string 型（ここではこの型は文字列型を表すものだと理解してください）のデータを記憶する場所を確保し、そこに yourname という名前をつけました。ここには後で値（キーボードから入力された文字列）が格納されます。このように実行中に値が変わり得るオブジェクトを変数と呼びます。ここでは yourname が変数です。

標準入力から文字列を読む

プログラム 2.9 の 7 行目は標準入力からの入力です。標準入力オブジェクトは std::cin という名前でシステムが用意しています。ここでの入力は、出力と同様に、やはり二項演算子である >> を使っています。

プログラム 2.10: 標準入力から読む

```
1  #include <iostream>
2  int main()
3  {
4          int a, b;
5          std::cin >> a >> b;
6          std::cout << a << '+' << b << '=' << a + b << std::endl;
7          return 0;
8  }
```

プログラム 2.11: 不正入力への対処

```
1  #include <iostream>
2  int main()
3  {
4          int a, b;
5          std::cin >> a >> b;
6          if(std::cin)
7                  std::cout << a << '+' << b << '=' << a + b << std::endl;
8          else
9                  std::cout << "Input error" << std::endl;
10         return 0;
11 }
```

2.5.2 標準入力からの整数の読み込み

プログラム 2.10 は標準入力から二つの整数値を読み込むようにしたものです。このプログラムを実行すると入力待ちになるので、整数を二つ、間をスペースで区切って入力します。最後に改行を入れると結果を表示します。改行はスペースと同じ扱いになりますので、整数を一つ入れたところで改行を入れて、次の整数を入れても構いません。

不正入力への対処

プログラム 2.10 は標準入力から二つの整数値が入力されることを想定しています。この時、整数値と空白（改行を含む）以外の文字が入力されたらどうなるでしょう。実際にアルファベットや小数点数や区切りをカンマにしてみるなどしてみてください。

数字や空白を入れても一応動くには動きますが意味の無い答えを出してきます。これでは困りますので、対処しましょう。対処の方法はいくつか考えられますが、一例として プログラム 2.11 を示します。

ここでは `if`-文による条件分岐を使っています。

2.6 言語規格とコンパイラ、インタープリタ

さて、プログラムを書いて実行するまでの手順を概観しましたが、C++ の言語規格が定めているのはプログラムの（文字による）記述の仕方と、それを実行した場合のコンピュータの動作を定めているだけです。つまり、実行に至る手順までも定めているわけではありません。また、実行そのものを言語処理系がやるのか否かも定めていません。前節で述べた方式では、実行ファイルができるだけで、実行そのものは OS の機能を使って実行しました。

2.6.1 コンパイラとインタープリタ

文字で記述されたプログラムを実行させるには、大きく分けて二つの方式があります。一つは前節で述べたように、コンピュータや OS が直接実行できるようにすべて機械語に変換してしまってから実行させる方式で、この変換する作業をコンパイル (compile)、それを行うプログラムのことをコンパイラ（compiler）と呼びます。もう一つは逐次解釈（interpret）しながら実行していく方式でこれを行うプログラムをインタープリタ（interpreter）と呼びます。

C++ の言語規格は、実行可能な形にする作業を翻訳またはトランスレート（translate）と呼び、どちらの方式で行うのかまでは規定していません。

一般にコンパイル方式の方が最適化を行いやすく、インタープリタ方式に比べ実行は高速になります。またメモリ等、計算機資源の利用効率もよくなります。一方、インタープリタ方式では逐次解釈実行していきますので、プログラムのエラー箇所の発見などが容易に行え、プログラムの開発効率が上がります。

C/C++ はもともと実行速度や計算機資源の利用効率を重視するような場面で多く使われてきた関係で、C/C++の言語処理系の多くはコンパイル方式が採られています。ここで説明に用いる g++ はその所属するプログラムグループの名前（GCC:GNU compiler collection) から容易に想像つくようにコンパイル方式のプログラムです。

2.6.2 コンパイラオプションの利用

通常コンパイラは様々なオプションを持っていて、コンパイルする時に、オプションを指定することができます。よく使うのは最適化の指定、デバッグモードの指定、警告レベルの指定などです。特に C++ の場合は、できるだけ翻訳時にエラーチェックを行うという思想で作られてますから、警告レベルは最大にして利用するのがよいでしょう。ここでは、g++ を例に、オプションを使ってみましょう。他のコンパイラでも、これらに該当するオプションは通常ありますので、マニュアルなどで調べてみてください。

警告レベルを最大に

ここで例示している g++ で hello.cpp に対してこれを行うには

```
g++ -Wall hello.cpp
```

と指定します。

出力ファイル名を与える

ここで使っている g++ では、出力ファイル名は指定しなければ Linux であれば a.out 、Windows であれば a.exe です。この名前は -o オプションで指定することができます。例えば Windows で上記プログラムの出力を hello.exe にしたい場合は

```
g++ -o hello.exe hello.cpp
```

とします。

前に説明した警告レベルの変更も付け加えるなら

```
g++ -Wall -o hello.exe hello.cpp
```

とします。

第3章　型と式、演算子

C++ プログラム言語では、処理の対象をオブジェクトと呼びますがその性質に応じて、扱いが異なります。この性質による分類を型（type）と呼びます。

変数や関数の戻り値などは使用する前に、それがどのような型であるのかを記述する必要があります。この記述を型宣言（type declaration）と呼びます。宣言のうち、実体を伴うものを定義と呼びます。

型には基本型（fundamental types）と複合型（compound types）があります。基本型には以下のものがあります。

- bool 型
- 文字型
- 整数型
- 浮動小数点数型

さらに情報が無いことを示す

- `void` 型

があります。

複合型としては

- 列挙型
- 配列型
- クラス（構造体）

があり、このうち、列挙型とクラスはユーザが定義を与えて使うのでユーザ定義型、残りは型を宣言するだけで使えるので組み込み型と呼ぶことがあります。

また、これらの格納域に関連して、

- ポインタ型
- 参照型

があります。

C++ の教科書などで、型の説明の中に POD 型という言葉が出てくることがあります。これは plain old data の略で、大まかに言うとプログラミング言語 C から持ってきた型で、そのオブジェクトがメモリ上に連続的に割り当てられるものです。

表 3.1: 整数型

型	用途
bool	真理値を表す。値は false か true。
char	文字を表す。符号付か否かは処理系異存。
signed char	符号付整数で文字を表す。
unsigned char	符号なし整数で文字を表す。
wchar_t	ワイド文字を表す。
short int	符号付の短い整数を表す。
unsigned short int	符号なしの短い整数を表す。
int	処理系に適した符号付整数を表す。
unsigned int	処理系に適した符号なし整数を表す。
long int	符号付の長い整数を表す。
unsigned long int	符号なしの長い整数を表す。

3.1 算術型

データ処理と言った時に、まず思い浮かぶのは数値データを処理することでしょう。数値データに対応するオブジェクトを算術型（arithmetic types）オブジェクトと呼びます。算術型は更に整数型と浮動小数点型に分類されます。前者は名前の通り整数に対応するものであり、後者は実数（の近似値）に対応するものです。真理値や文字も C++ では整数型に分類されます。どちらもある一定範囲の整数に一対一で対応させることができるからです。

算術型オブジェクトはコンピュータの記憶域に固定のビット数を使って格納されます。固定のビット数のデジタル表現では、有限の個数のデータしか表すことができません。従って整数であれば扱える整数の範囲が制限されます。また実数の場合は扱える範囲だけでなく精度も限定されます。これらのオブジェクトに割り当てるビット数を多くすれば扱える範囲を広くしたり、より高精度にすることができますが、記憶域をより多く必要とします。また場合によっては計算時間もより多くなります。このため C++ では整数や実数を表す型をそれぞれ複数用意しています。

3.2 整数型

表 3.1 に整数として扱われる型を示します。これらを総称して汎整数型（integral type）とか整数型（integer types）と呼びます。

整数型のうち int 型は、そのコンピュータで整数を表す自然な大きさを占めます。自然な大きさというのは、そのコンピュータにとってハードウェア的に効率のよい大きさです。より広い範囲を扱うには、より多くの記憶域が必要になり、より狭い範囲であればより少ない記憶域で済む可能性があります。ユーザにより、これらの指示ができるように、より広い範囲が扱える可能性のある

```
long int n;
```

及び、より狭い範囲での利用を可能とする

```
short int n;
```

があります。この場合 int は省略でき（実際、普通は省略します）

```
long n;
```

とか

```
short n;
```

のように書くことができます。ただし、範囲の大きさが実際にどれだけであるかはシステム依存です。規格上は、例えば long int は int より範囲が狭くてはならないというだけで必ずしも扱える範囲が広がるわけではありません。

これらには、いずれも正負の符号付きと符号無しの区別があります。正負の符号付きとする場合は、例えば

```
signed long int n;
```

符号無しの場合は、例えば

```
unsigned short int n;
```

のように書きます。省略した場合は符号付きです。この場合も int を省略して、

```
signed long n;
```

とか

```
unsigned short n;
```

のように書くこともできます。

符号付き整数の各々の型に対応する符号なし整数の型とは同じ大きさの記憶域を占めます。例えば short int が 16bit であれば、unsigned short int も 16bit です。非負数に対しては、符号付き整数の表現と、それに対応する符号なし整数の表現は一致します。

同じ型の符号付と符号なしは記憶域で同じサイズを占めます。例えば、

```
char    signed char    unsigned char
```

の記憶域に占めるサイズは同じです。つまり、char が記憶域で 8 ビットを占めるならば、signed char も unsigned char も 8 ビットを占めます。同様に short int と unsigned short int も同じサイズです。更に int と unsigned int も long int と unsigned long int も同じサイズです。

```
short int    unsigned short int
```

の記憶域に占めるサイズは同じですし、

```
int    unsigned int
```

の二つも、同じサイズです。

3.2.1 bool 型

bool 型（ブール型）というのは二つの値、true（真） と falase（偽）をとるものです。真理値型とか真偽値型と呼ばれることもあります。

表 3.2: 整数型が扱える最小範囲

型	最小値	最大値
signed char	-127	+127
unsigned char	0	255
short	-32767	+32767
unsigned short	0	65535
long	-2147483647	+2147483647
unsigned long	0	4294967295

プログラム 3.1: 整数型の範囲の表示

```
1  #include <iostream>
2  #include <climits>
3
4  int main()
5  {
6          std::cout
7                  << "␣char␣[" << CHAR_MIN << "," << CHAR_MAX << "]\n"
8                  << "short␣[" << SHRT_MIN << "," << SHRT_MAX << "]\n"
9                  << "␣␣int␣[" <<  INT_MIN << "," <<  INT_MAX << "]\n"
10                 << "␣long␣[" << LONG_MIN << "," << LONG_MAX << "]\n";
11         return 0;
12 }
```

3.2.2 文字と整数

文字は、個々の文字の図形（図形文字の場合）や機能（制御文字の場合）に整数を割り振ることで表現されます。少なくとも実行時の任意の基本文字集合に含まれる文字を表現できる型として char が用意されています。この型は signed char 型や unsigned char 型とは独立した型として扱われますが、この型の値の取り得る範囲は、signed char 型か unsigned char 型のどちらか一方に一致します。どちらに一致するかは実装依存です。

この char 型、signed char 型および unsigned char 型の記憶域の大きさは同じです。この記憶域の大きさをバイトと呼びます。多くのシステムでは 8 ビットの記憶域を割り当てています。つまり、256 個より多くの種類の文字を扱うことができません。これを解決するために多バイトで一文字を表現するなどの方法がとられますが、その場合は、文字比較など文字として用意されている言語系の機能が使えなくなります。このため扱えるすべての拡張文字を表現できる型として wchar_t 型が用意されています。

3.2.3 整数型が扱える範囲

整数型の扱える数値の範囲は処理系依存です。これでは実際問題を扱う場合に困るので、規格では最小範囲を定めています。これより大きい値の範囲であることもあります。

処理系での整数型の範囲などは、<limits> または <climits> を利用することで取得できます。後者は C 言語から引き継いでいるものです。前者はテンプレートの知識を必要とするので、ここでは <climits> を使って範囲を調べる例をプログラム 3.1 に示します。

表 3.3: 浮動小数点数型の範囲・精度の最小値

型	最小値	最大値	10 進精度	分解能 ϵ
`float`	1×10^{-37}	$1 \times 10^{+37}$	6 桁	1×10^{-5}
`double`	1×10^{-37}	$1 \times 10^{+37}$	10 桁	1×10^{-9}
`long double`	1×10^{-37}	$1 \times 10^{+37}$	10 桁	1×10^{-9}

3.2.4 整数型の昇格

整数型に分類される各型は、より範囲の広い型に変換できます。これを昇格（promotion）と言います。同じ型の符号付きと符号無しとでは符号無しの方が広いとされます。

3.2.5 真理値型の昇格

真理値型が整数型の他の型に昇格すると、`true` は 1 に、`false` は 0 に変換されます。

3.3 浮動小数点数型

私たちの身の回りには、整数だけでなく実数計算も数多く存在します。整数型が扱える範囲は非常に限られているので、実数も扱えると便利です。ところが実数は連続なので、有限のビット数では厳密には表せない場合がほとんどです。しかし、現実の世界では、近似値で十分である場合もあります。浮動小数点数というのは、実数の（デジタル計算機上での）近似値表現です。

近似表現に際しては、仮数部と指数部にそれぞれ固定の有限ビット数を割り当てます。整数よりは遥かに広い範囲が扱えますが、扱える範囲には（整数型と同様）制限があります。また近似値ですので、精度が問題になります。今、変数 `r` が浮動小数点数型のデータ変数であることを示す場合、この記憶域の大きさにより

```
float r;
```

及びより広い範囲、より高精度での利用を可能とする

```
double r;
```

更により広い範囲、より高精度での利用を可能とする

```
long double r;
```

があります。ただし、整数の `long` 同様に、同等か若しくはそれ以上ということであって、処理系に依存します。どの処理系であっても保証される範囲や精度を表 3.3 に示します。

処理系に依存するこれらの値を調べるには、ヘッダ `<limits>` か `<cfloat>` を用います。前者の `<limits>` を使うにはテンプレートの知識が必要ですので、ここでは `<cfloat>` を使う例をプログラム 3.2 に示します。この `<cfloat>` ヘッダは C 言語の `<float>` ヘッダに対応するものです。

プログラム 3.2: 浮動小数点型の限界値の表示

```
1  #include <iostream>
2  #include <cfloat>
3  int main()
4  {
5          std::cout << "      float ["
6                    <<  FLT_MIN << "," <<  FLT_MAX << "] "
7                    <<  FLT_EPSILON << "\n";
8          std::cout << "      double ["
9                    <<  DBL_MIN << "," <<  DBL_MAX << "] "
10                   <<  DBL_EPSILON << "\n";
11         std::cout << "long double ["
12                   << LDBL_MIN << "," << LDBL_MAX << "] "
13                   << LDBL_EPSILON << "\n";
14         std::cout << std::flush;
15         return 0;
16 }
```

3.4 型宣言

変数 n の型を宣言するには、

```
        型 n;
```

と書きます。例えば n が int 型であることを宣言するには

```
        int n;
```

と書きます。

3.4.1 const と volatile

変数の値を変えないことを宣言するには const 修飾子（const qualifier）を付与します。例えば

```
        const int n = 3;
```

と宣言（この場合は定義）すると、変数 n は初期化以外、値を変えることはできなくなります。逆に言えば宣言が定義を兼ねている場合は、必ず初期化が必要になります。

一方、オブジェクトの中には、例えば通信や外部機器に接続されているものなど、処理系が検知できない要因で値が変わる可能性のあるものがあります。これらは値が変わっていないとして最適化されると困る場合があります。処理系に処理系が検知できない要因で値が変わる可能性があることを宣言するには volatile 修飾子（volatile qualifier）を使います。

この const 修飾子と volatile 修飾子の二つの修飾子をまとめて cv 修飾子とと呼ぶことがあります。

表 3.4: エスケープシーケンスによる文字表記

改行	NL	`\n`
水平タブ	HT	`\t`
垂直タブ	VT	`\v`
バックスペース	BS	`\b`
復帰	CR	`\r`
改頁	FF	`\f`
警報（ベル）	BEL	`\a`
バックスラッシュ	`\`	`\\`
疑問符	`?`	`\?`
単一引用符	`'`	`\'`
二重引用符	`"`	`\"`
8 進数	*ooo*	`\`*ooo*
16 進数	*hhh*	`\x`*hhh*

3.5 リテラル

リテラルというのは、すでに 2.3 でも述べたように文字による値の表記法です。値を直接書きますので、プログラムの実行時に変更されることはありません。つまり定数です。実際プログラミング言語 C の規格では、これを定数（constant）と言います。ところが定数というのは他の場面でも色々使います。特に C++ では constant を表す `const` というキーワードを様々な状況で使います。そこで、値を直接文字表記したものの方はリテラルと呼んでいます。値の書き方はデータの型によって異なります。以下では型ごとに書き方を見てみましょう。

3.5.1 文字リテラル

文字は通常（いわゆる半角英数文字を）単一引用符 ' で囲んで表現します。例えば `'A'` はアルファベットの第一文字である `A` という文字を表し、`'+'` は記号文字 `+` を表します。ただし、改行などの制御文字（見えない文字)、単一引用符 ' およびバックスラッシュ \ はこの記法では書けません。表 3.4 にこのような文字の表記法を示します。表の一番右の欄が表記法です。このうち、疑問符と二重引用符はエスケープシーケンスを使わずに直接書くこともできます。8 進数と 16 進数による表記では文字コードを表します。表 3.4 では 3 桁で指定してありますが、1 桁、2 桁でも構いません。例えば ASCII 文字セットでは、大文字の `A` のコードは 10 進数で `65` ですので、ASCII 文字セットを使っている場合、

```
'A'    '\x41'    '\101'
```

は、いずれも大文字の `A` を表します。これを確かめるためプログラム 3.3 では上記 3 つの文字表現を使って、間に水平タブを挟んで表示しています。

プログラム 3.3: 同一文字の異なる表現

```
1 #include <iostream>
2 int main()
3 {
4         std::cout << 'A' << '\t' << '\x41' << '\t' << '\101' << std::endl;
5         return 0;
6 }
```

プログラム 3.4: 長い文字列の分割表現

```
1 #include <iostream>
2 int main()
3 {
4         std::cout <<
5                 "High Energy " "Accelerator "
6                 "Research Organization"
7                 << std::endl;
8         return 0;
9 }
```

3.5.2 文字列リテラル

文字列リテラルはすでにプログラム 2.2 で`"hello, world"` として出てきました。このように文字の並びを二重引用符でくくったものです。ただし二重引用符とバックスラッシュはこの記法では書けません。文字リテラルと同様に表 3.4 に示すエスケープシーケンスによる記法で書きます。単一引用符は二重引用符の中では、それ自身で表すこともできますし、エスケープシーケンスを使って表すこともできます。

連続する文字列は翻訳の過程で結合され一つの文字列にされます。つまり文字列が長い場合、複数の文字列に分けて書くことができます。例えば

```
        "High Energy Accelerator Research Organization"
```

は

```
        "High Energy " "Accelerator "
        "Research Organization"
```

と 3 つの文字列に分けて書いても全く同じ単一の文字列オブジェクトが生成されます。空白も二重引用符の中に置かれていることに注意してください。改行は空白文字と同じ扱いなので、上の例のように文字列を複数の行に置くことができます。ただし二重引用符の中に改行を含めることはできません。エラーになります。

プログラム 3.4 は、上の例を具体的にプログラムにしたものです。

3.5.3 真理値リテラル

真理値というのは真か偽かの二つの値を持つものです。C++ でこの値を表現したい場合は `true`（真の場合）または `false`（偽の場合）と書きます。二つとも予約語です。

3.5.4 整数リテラル

整数リテラルは 2.3 でも見たように、10 進数、8 進数、16 進数による表記法があります。

10 進数リテラル 10 進数リテラルは、1 から 9 までのいずれの数字から始まり 0 から 9 までの数字が続くものです。

8 進数リテラル 8 進数リテラルは、数字 0 で始まり、0 から 7 までの数字が続くものです。

16 進数リテラル 16 進数リテラルは、0x または 0X で始まり、16 進数字（すなわち、0 から 9 までの数字、A から F までの文字、または a から f までの文字のいずれか）が続くものです。

いずれも末尾に L または l をつけることで long であることを、また U または u をつけることで unsigned であることを示します。また例えば UL のように、これらを両方つけることで unsigned long であることを示します。

3.5.5 浮動小数点数リテラル

浮動小数点リテラルは、整数部、小数点、小数部、指数部から成り、整数部または小数部のいずれかと小数点または指数部のいずれかは省略することができます。指数部は E または e で開始されます。指数部は符号をつけることができます。

末尾に F または f をつけることで float であること、また L または l をつけることで long double であることを示します。これらがついていなければ double です。

3.6 型変換

算術演算を行なう場合に型の変換を行なう必要が生じる場合があります。例えば二項演算子 + の二つのオペランドの型は一致する必要がありますが、片方が異なる場合はどちらかを変換しなければなりません。多くの場合、処理系はこれを自動で行います。また処理系にまかせずに、明示的に指示することもできます。ここでは、これらについて述べます。

3.6.1 型の昇格

型の昇格とは、算術型の値を、値を維持したまま、より広い範囲の型に変換することを言います。

整数の昇格 char、signed char、unsigned char、short および unsigned short 型の値は、それらの型の値すべてを int 型で表すことができるならば int 型に、そうでなければ unsigned int 型に変換できます。

列挙型の昇格 列挙型の値は、int、unsigned int、long および unsigned long の順に、そのとり得る値すべてを含む最初の型に変換できます。

真理値型の昇格 真理値型の値は int 型に変換できます。このとき true は 1 に、false は 0 に変換されます。

浮動小数点数型の昇格 float 型の値は double 型に変換できます。

3.6.2 算術変換

処理系に組み込まれている二項演算子のうち、算術演算を行なう演算子と比較演算を行なう演算子の二つのオペランドの型は一致している必要があります。これが一致していない場合、どちらかの型が変換されます。演算後の型はオペランドの型になります。この時の変換は、原則的には、より広い範囲を扱える型の方に変換されます。具体的には、次の規則が順に適用されます。

1. 一方のオペランドの型が long double ならば、もう片方も long double に変換されます。
2. そうではない場合、一方のオペランドの型が double ならば、もう片方も double に変換されます。
3. そうではない場合、一方のオペランドの型が float ならば、もう片方も float に変換されます。
4. そうではない場合、整数の昇格が実行され、次のステップに進みます。
5. 整数の昇格が行なわれた後、一方のオペランドの型が unsigned long ならば、もう片方も unsigned long に変換されます。
6. そうではない場合、一方のオペランドの型が long で、もう片方が unsigned int ならば
   (a) unsigned int のすべての値を long に収容できるならば long に変換され
   (b) そうでない場合は、両方のオペランドとも unsigned long に変換されます。
7. そうではない場合、一方のオペランドの型が long ならば、もう片方も long に変換されます。
8. そうではない場合、整数昇格のまま、すなわち両方のオペランドともに int です。

プログラム 3.5: 型変換の記法

```
1  #include <iostream>
2  int main()
3  {
4          double pi = 3.1415926536;
5          int i, j, k;
6          i = (int)pi;
7          j = int(pi);
8          k = static_cast<int>(pi);
9          std::cout << i << ',' << j << ',' << k << std::endl;
10 }
```

3.6.3 暗黙の数値変換

暗黙の数値変換は、代入や初期化、関数の引数への引渡しなどで算術型の値が別の算術型の値に変換されることを言います。基本的には、算術型から別の算術型への変換は常に可能です。ただしこの場合、変換によって情報が失われることがあります。例えば浮動小数点数を整数型に代入すると、小数部は 0 方向に切り捨てられます。また、変換後の型が元の値を保持できない場合もあり、このときの振る舞いは不定です。

3.6.4 明示的型変換

型変換は明示的に指示することもできます。これをキャスト（cast）と呼びます。これには、いくつかの方法があって、

1. 型の名前を括弧 () で囲み式の前につける。これは C 言語の方式を受け継いだものです。
2. 型の名前を関数のように扱い、型名の後、変換する式を括弧 () で囲む。
3. 用途により次のどれかに続けて、型名を <> で囲み、変換する式を () で囲む。
   - `const_cast`
   - `dynamic_cast`
   - `reinterpret_cast`
   - `static_cast`

このうち最後の用途による区別は、今の時点で使う可能性があるのは `static_cast` だけです。算術変換や逆方向の算術変換（例えば `long` から `short` への変換）などはこれで行います。例を示します。プログラム 3.5 では 3 通りの方法で `double` から `int` への変換が行なわれています。いずれも `double` から `int` への変換を指示し、これを実行すると、

```
3,3,3
```

という結果が得られます。

3.7 式と演算子

演算子は、一つ、または複数のオブジェクトに作用して、ある型の値を生み出したり、オブジェクトの値や状態を変化させたりします。ある一つの演算子が関与するオブジェクトをその演算子のオペランドと呼びます。またオブジェクトや演算子の組み合わせにより作用を記述したものを式と呼びます。一般に式は、ある型の値を生成しますが、何も生成しない場合もあります。ここでは算術型オブジェクトをオペランドに持つ演算子について述べます。

多くの演算子はオーバーロード（上書き）して、その機能を変えることができます。ここで述べるのは、処理系に最初から組み込まれている役割です。

3.7.1 増分と減分

増分演算子 ++ と減分演算子 -- は対象となる算術オブジェクトの値を 1 だけ加えたり（増分演算子の場合)、減じたり（減分演算子の場合）します。

```
int i, j;
i = j = 5;
++i; --j;
```

とすると、i は 6 に、j は 4 になります。

これらには、前置と後置があって、前置の場合は式の値を評価する前に増減を行い、後置の場合は式の値を評価した後で増減を行います。

```
int i, j, k, l;
i = j = 5;
k = ++i;
l = j++;
```

では、k は = の右オペランドの値は、値を評価する前に i の増分が行なわれるので、6 になりますが、l の= の右オペランドの式の値は j の増分が行なわれる前の値ですので、5 になります。

なお、後置の増分、減分は前置の増分、減分よりも優先度が高いことに注意します。

論理値型に対する増分と減分

論理値（bool）型に対しては（前置、後置ともに）増分のみが認められています（演算結果オブジェクトの新しい値は必ず true になる）が、推奨されていません。論理値型に対して増分、減分は使わない方がよいでしょう。

3.7.2 補数

演算子 ~ は、一つの整数型または列挙型オブジェクトを右オペランドとし、ビット単位で補数をとります。つまりビット単位で 0 と 1 が入れ替わります。記号 ~ の代わりに予約語 compl を使うこともできます。

3.7.3 論理否定

演算子 ! は、オペランドを bool 型に変換した後、その論理否定を返します。結果は bool 型になります。記号 ! の代わりに not を使うこともできます。

3.7.4 正負符号

正符号 + は値そのものを返すだけですので、通常は余り使いません。負符号 - は符号を反転します。符号無しの型に対して作用させた場合は、2^n （n はビット数）から元の値を引いた値、すなわち 2 の補数になります。

3.7.5 乗除算

乗算に対する演算子*および除算に対する演算子/は二項演算子で、両方のオペランドに算術型または列挙型をとります。左右のオペランドの型が異なる場合は、通常の型変換を行なって型を揃えた後、演算が行なわれます。整数に対する除算では小数部は切り捨てられます。剰余算に対する演算子%も二項演算子で、両方のオペランドは整数型または列挙型でなければなりません。

第二オペランドがゼロである場合の除算および剰余算の振る舞いは不定です。

結果が、その型で表現できる最大値を越える場合、符号無し整数に対する演算では、その型で表現できる最大値を法とした演算が行なわれます。それ以外の型の場合、振る舞いは不定です。

3.7.6 加減算

加算を行なう演算子 + および減算を行なう演算子 - はいずれも二項演算子で、両方のオペランドに算術型または列挙型をとります。左右のオペランドの型が異なる場合は、通常の型変換を行なって型を揃えた後、演算が行なわれます。

3.7.7 シフト演算

シフト演算子には左シフト演算子 << と右シフト演算子 >> があり、いずれも二項演算子です。両方のオペランドは整数型か列挙型で整数型に昇格の後、ビットシフト演算が行なわれます。いずれも第 2 オペランド（右オペランド）がシフトさせるビット数で、第 1 オペランド（左オペランド）がシフトされます。第 2 オペランド（右オペランド）が負の場合や、第 1 オペランド（左オペランド）よりビット数が多い場合は、結果は不定です。

左シフトの場合、空いたビットは 0（ゼロ）で埋められます。右シフトの場合は、第 1 オペランド（左オペランド）が符号付きの正の値または 0 ゼロの場合、あるいは符号無し整数の場合には、空いたビットは 0 で埋められます。

3.7.8 関係演算

値の比較を行い結果は bool 型の値になります。これも二項演算子です。

`a < b`	a が b よりも小さければ `true`、そうでなければ `false`
`a > b`	a が b よりも大きければ `true`、そうでなければ `false`
`a <= b`	a が b 以下であれば `true`、そうでなければ `false`
`a >= b`	a が b 以上であれば `ture`、そうでなければ `false`

表 3.5: 算術演算を伴う代入演算子

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

3.7.9 等価演算

値の比較をして等しいかどうかを bool 型の値で返します。二項演算子です。

a == b	a と b が等しければ true、そうでなければ false
a != b	a と b が等しくなければ true、そうでなければ false

3.7.10 ビット単位の論理演算

ビット単位の論理演算には、ビット積&、ビット差（排他的論理和）^ およびビット和|があります。いずれも二項演算子で、オペランドは整数型のみです。

3.7.11 論理積と論理和

論理積 && および論理和 || も二項演算子で、両方のオペランドを bool 型として扱い結果を bool 型で返します。ただし、この二つの演算では短絡評価が行なわれます。つまり第 1 オペランドが先に評価され、その時点で、式の結果が確定した場合、第 2 オペランドは評価されません。

3.7.12 二択条件演算

三項演算です。

```
条件 ? 式1 : 式2
```

という形式です。第 1 オペランド（条件）は bool 型に変換され、その値が true であれば第 2 オペランド（式 1）が、false であれば第 3 オペランド（式 2）が評価されます。

3.7.13 代入

代入演算子 = は右オペランドを評価しその値を左オペランドに代入します。この演算子の他に、表 3.5 に示すような op= という形式の演算子も用意されています。これは演算 op を伴う代入で

```
x op= y;
```

は意味的には

```
x = x op y;
```

です。しかし前者は x の評価は一度だけであることに注意します。いずれも結果の型と値は左オペランドになります。

3.7.14　カンマ式

カンマ演算子 , は左オペランドを評価して結果を破棄し、右オペランドを評価します。結果は右オペランドの値であり、結果の型は右オペランドの型になります。

3.8 関数

一連の手続きに名前をつけ、名前による呼び出しだけで値を得られるようにしたものを関数と呼びます。関数は

```
戻り値の型 識別名 (引数リスト)
```

という構文により宣言され、

```
戻り値の型 識別名 (引数リスト) { 本体 }
```

という構文により定義されます。

戻り値の型は `int` 、`double` などの型名を書きます。値を何も返さない関数というのも許されます。その場合、戻り値は `void` と指定します。

識別名は、関数の名前で、変数などの識別名と同じ規則で名前を付けます。

引数リストには、型名と（仮の）変数名の対を引数の数だけカンマ（`,`）で区切って並べます。宣言の場合は型名だけでも構いません。引数が無い場合、引数リストは空にするか、`void` と指定します。

戻り値が `void` の場合を除いて、本体には `return` 文が必ず存在し、戻り値の型の値を戻します。

3.9 数値計算の注意事項

整数や浮動小数点数を扱う上での、いくつかの注意事項を述べておきます。多くの場合、整数や浮動小数点数は、日常使っている整数や実数（の近似値）として問題ありませんが、条件によっては予想もしなかった結果になることがあります。例を示しながら、これらの問題を見てみましょう。

なお、ここで示す問題は言語系の問題というよりは、デジタル表現に伴う問題ですので、他の言語であっても同様な問題があります。

3.9.1 整数同士の除算に伴う問題

整数同士の除算の結果は整数で、端数は切り捨てられます。従って、整数同士の掛け算と割り算の順序は一般に入れ替えることはできません。例えば

```
        int a(3), b(7), c(2), d;
        d = a * b / c;
```

と

```
        int a(3), b(7), c(2), d;
        d = a / c * b;
```

は異なる結果になります。前者は掛け算した結果 21 を 2 で割って結果 10（端数切捨て）になりますが、後者は 3 を 2 で割ってその結果（端数切捨てなので 1 になります）に 7 を掛けますから結果は 7 です。

3.9.2 型変換に関する誤解

例えば二項演算子の一方のオペランドが `float` でもう一方のオペランドが `int` の場合には、`int` の方のオペランドを `float` に変換して演算を行うので、

```
#include <iostream>
int main()
{
        int a = 3;
        int b = 2;
        float f = 1.0 + a / b;
        std::cout << f << std::endl;
        return 0;
}
```

とすると `f` は 2.5 になるかというと、そうはなりません。なぜなら、加算より先に `a / b` が実行されますが、両オペランドともに `int` 型なので、型変換は行なわれずに整数同士の割り算が実行されます。つまり結果は整数値 1 になり、その後、加算を実行しようとした時、`flot` 型と `int` 型の加算なので、ここではじめて `int` 型から `float` 型の変換が起こります。つまり、端数切り捨てが行なわれた後で、`int` から `float` の変換が行なわれます。

もし `f` の値が 2.5 になるようにしたいのであれば、最初から `a, b` を `float` にしておくか、あるいは

```
        float f = 1.0 + float(a) / float(b);
```

のように割り算の前に強制的に型変換を行います。

3.9.3 上位ビットの切り捨て

次のプログラムは処理系依存です。多くの処理系は unsigned char は 8 ビットですが、もしそうなら、次のプログラムで n は 1027 にはなりません。筆者の環境ではエラーも警告も無しに 3 になりました。

```
#include <iostream>
int main()
{
        unsigned char uc = 1026;
        int n = uc + 1;
        std::cout << n << std::endl;
        return 0;
}
```

この問題の原因は、

```
        unsigned char uc = 1026;
```

にあります。1026 は 8 ビットでは表せません。unsigned に対するルールとして、格納するのに十分なビット数が無い場合には上位ビットが捨てられます。

3.9.4 符号無しと符号付き整数の混在

次のプログラムも処理系依存ですが、一般に c の値はとても大きな値になるでしょう。

```
#include <iostream>
int main()
{
        int a = -3;
        unsigned int b = 2;
        int c = a / b;
        std::cout << c << std::endl;
        return 0;
}
```

符号無し整数と、符号付き整数とでは符号無し整数の方が扱える範囲が広いとされます。従って二項演算子のオペランドに符号無し整数と符号付整数がくると、符号無し整数に変換されてから演算が行なわれます。符号ビットは一般に最上位ビットですので、符号無しに変換すると非常に大きな値になります。

では、割り算ではなく足し算ではどうでしょう？

```
        int c = a + b;
```

とした場合です。これも結果は処理系に依存しますが、多くの場合、正しい答え -1 が得られるでしょう。これは代入先が int で最上位ビットが再び符号ビットとして解釈されることになるからです。割り算の場合は、符号無し整数同士の割り算の結果、上位ビットは 0 になって int に戻しても符号は復活されなかったからです。

3.9.5 浮動小数点数の精度不足

次の例も処理系依存です。これも一般に結果は 0 にはなりません。つまり、if 文や while 文の条件部に、この f - 1234567890.0 のような書き方がされていると、処理系によっては常に真である可能性があります。

```
#include <iostream>
int main()
{
        float f = 1234567890.0;
        std::cout << f - 1234567890.0 << std::endl;
        return 0;
}
```

これが 0 にならないのは float に 10 桁の精度が無い場合に生じます。浮動小数点数を扱う場合には、精度に対する十分な考察が必要です。

3.9.6 精度不足による無限ループ

浮動小数点数の精度が問題になる例をもう一つ挙げます。以下のプログラムは、きちんと FLT_EPSIRON により精度を得て、その 2 倍の値と 10 倍の値を使っているので精度的には問題が無いように見えます。変数 b は変数 a に FLT_EPSILON の 10 倍を加えたものです。一方、a は繰り返しの中で delta ずつ、すなわち FLT_EPSILON の 2 倍ずつ増加しています。つまり繰り返しを 5 回行えば終了するはずです。

```
#include <iostream>
#include <cfloat>
int main()
{
        float delta = FLT_EPSILON * 2.0;
        float a = 1.1;
        float b = a + (FLT_EPSILON * 10.0);
        int n = 0;
        while (a < b)
        {
                a += delta;
                ++n;
        }
        std::cout << n << std::endl;
        return 0;
}
```

ところが、これも処理系に依存しますが、無限ループに陥る可能性があります。実際、筆者の環境では無限ループに陥ります。これは、delta の値が a に比べて非常に小さい場合、精度が保てなくなるからです。たとえ最初に a に対して精度が保てていても、a が大きくなるにつれ、delta の相対値はどんどん小さくなっていくために精度が保てなくなります。

これを改善する一例を示しておきます。

```
#include <iostream>
#include <cfloat>
int main()
{
        float delta = FLT_EPSILON * 2.0;
```

```
        float a0 = 1.1;
        float b = a0 + (FLT_EPSILON * 10.0);
        int n = 0;
        float a = a0;
        while (a < b)
        {
                ++n;
                a = a0 + (delta * n);
        }
        std::cout << n << std::endl;
        return 0;
}
```

このプログラムでは、増加分を `(delta * n)` として毎回計算しています。これにより増加分が相対的に小さくなることを防いでいます。

第4章　配列とポインタ、参照

C++ が想定する計算機の記憶域にはアドレス（address、番地）がつけられています。その記憶域の内容の読み出しや書き込みは、すべてこのアドレスを通じて行われます。更にアドレスは char 型のオブジェクト 1 個が記憶域に占める大きさを 1 とする単位、すなわちバイト単位でつけられているものとしています。C++ で言うバイトとは、char 型のオブジェクト 1 個が記憶域に占める大きさのことであって必ずしも 8 ビットではありません。

番地を使ってプログラムがどのように実行されるか見てみましょう。例えば

```
int a, b;
a = 3;
b = a + 2;
```

と書かれている時、まず 1 行目で整数値を格納する記憶域が二つ確保されますが、その番地が 1020 番地と 1060 番地だったとしましょう。2 行目の意味は 1020 番地に整数値 3 を代入せよとなります。同様に 3 行目の意味は 1020 番地の値を読み出し、その値に整数値 2 を加え、結果を 1060 番地に代入せよとなります。このように、変数名は「その名前に割り当てられた番地の内容」という意味です。

4.1　ポインタ

C++ では、他のオブジェクトに割り当てられた番地を内容とするオブジェクトも扱うことができます。このオブジェクトをポインタ（pointer）型オブジェクトと呼びます。ポインタ型オブジェクトを宣言するには、その番地の指し示す先にあるオブジェクトの型も指定します。例えば整数型のオブジェクトを格納している番地を内容とする変数 p を宣言するには

```
int * p;
```

のように書きます。同様に float 型のオブジェクトへの番地を格納する変数を宣言するには、

```
float * p;
```

のように書きます。このように、ある型のオブジェクトの番地を格納する変数を宣言するには型名の後に * を付けて書きます。

この * は、これ自身で区切り文字ですので、

```
int * p;
```

は

```
int* p;
```

と書いても、

```
int *p;
```

プログラム 4.1: ポインタを通じた操作

```
1 #include <iostream>
2 int main()
3 {
4         int a = 3;
5         int * p = & a;
6         *p = *p + 2;
7         std::cout << a << std::endl;
8         return 0;
9 }
```

と書いても、あるいは

```
        int*p;
```

と書いても構いません。

なお、ポインタの宣言は変数ごとに行われることに注意します。

```
        int * p, q;
```

と書いたとき、p は整数へのポインタですが、q は整数です。両方ともポインタと宣言するには、

```
        int * p, * q;
```

とします。

4.1.1 オブジェクトの番地を得る演算子

オブジェクトの番地を得る演算子は単項演算子 & です。例えば

```
        int a = 3;
        int * p = &a;
```

と書くと、ポインタ型変数 p は整数型変数 a の番地を初期値とします。

4.1.2 ポインタ型変数を通じた読み書き

ポインタ型変数の指し示す先を読み書きするには、単項演算子 * を使います。

```
        int a = 3;
        int* p = &a;
        int b = *p + 2;
```

とすると、整数型変数 b には p に格納されている番地の先の値、すなわち a の値に整数値 2 を加えたもの、つまり 5 が代入されます。また

```
        int a;
        int* p = &a;
        *p = 2;
```

とすると、p に格納されている番地の先の値、すなわち a の値として整数値 2 を代入します。プログラム 4.1 に、これらの機能を使った例を示します。見かけ上、変数 a への代入はどこにもありませんが、ポインタ型変数 p を通して、a から値を読み出し、2 を加え、その演算結果である 5 を a に代入していることに注意してください。

4.2 配列

配列は同一型のオブジェクトを記憶域上で連続した番地に並べたものです。例えば int 型のオブジェクトを 5 個格納するための記憶域を確保し、その識別名を a とするには

```
int a[5];
```

と書きます。同様に double 型のオブジェクトを 7 個格納するための記憶域を確保し、その識別名を r とするには

```
double r[7];
```

と書きます。

配列で重要なことは、単に指定された個数のオブジェクトを格納する記憶域が確保されるだけでなく、記憶域上で、これらのオブジェクトが連続した番地に配置されることです。

4.2.1 配列の初期化

配列を初期化するには、等号 = の後に、要素の値を句点 , で区切って並べ、全体を中括弧 {} で囲みます。例えば、

```
int a[5] = { 5, 4, 3, 2, 1};
```

と書くと、5 個の整数を格納するための記憶域が確保され、a[0] は 5 に、a[1] は 4 に、、と初期化されます。一番最後の要素の後に , があっても構いません。例えば上の例は、

```
int a[5] = { 5, 4, 3, 2, 1,};
```

と書くこともできます。もし、後で配列の大きさを変える可能性があるなら、, をつけておく方が間違いが少ないかも知れません。

4.2.2 配列要素へのアクセス

配列の個々の要素は配列変数名に、添え字（index）を [] でくくって指定することでアクセスできます。この時、先頭（すなわち一番目）の添え字は 0 であることに注意します。例えば上の例で、

```
a[1] = 5;
a[2] = a[1] + 2;
```

と書くと、2 番目の要素に整数 5 が代入され、3 番目の要素には 2 番目の要素の値に 2 を足した値、すなわち 7 が代入されます。

プログラム 4.2 は初期化した後、標準出力に各要素の値を表示するプログラムです。

4.3 配列とポインタ

配列は記憶域上で連続した番地に各要素が配列されます。従って配列の先頭位置の番地から、他の要素の番地も決定できます。このことから、配列の要素への読み書きには一般的にポインタが多用されます。

プログラム 4.2: 配列の初期化と要素の表示

```
1 #include <iostream>
2 int main()
3 {
4         int a[5] = { 5, 4, 3, 2, 1};
5         for (int i = 0; i < 5; i++)
6                 std::cout << "a[" << i << "] = " << a[i] << std::endl;
7         return 0;
8 }
```

4.3.1 記憶域上で占める大きさを求める演算子

オブジェクトの記憶域上で占める大きさは、型によりまちまちで、かつ処理系依存です。番地の計算では、オブジェクトの記憶域上で占める大きさが問題になることがあります。このため、オブジェクトの記憶域上で占める大きさを得る単項演算子 `sizeof` が用意されています。

オブジェクト `a` の記憶域上で占める大きさは `sizeof a` として得ることができます。戻り値の単位はバイトです。被演算子は `()` でくくっても構いません。つまり `sizeof a` は `sizeof(a)` と書くことができます。被演算子にはオブジェクトの識別名だけでなく、型名を書くこともできます。この場合、被演算子には `()` を必ずつけなければなりません。例えば `double` 型のオブジェクトの記憶域上で占める大きさは `sizeof(double)` で与えられます。定義により `sizeof(char)` は、どのような処理系であれ 1 です。

4.3.2 ポインタに対する演算

ポインタに対し、整数の加減算が可能です。ただし、この場合の加減算はポインタに格納されている値に整数値をそのまま加減算するのではなく、ポインタが指し示す先の型の大きさを単位とする加減算が行われます。つまり、配列の場合の添え字と同じ意味になります。

例を示します。

```
        int a[5] = { 5, 4, 3, 2, 1};
        int* p = &a[0];
        int b = *(p + 3);
```

の最後の行は

```
        int b = a[3];
```

と同じ結果になります。

同様に、ポインタ型オブジェクトに対しては単項演算子 `++` や `--` も適用可能です。この場合も増減値の単位はポインタが指し示す先の型の大きさになります。つまり `++` は次の要素の番地に、`--` は前の要素の番地になります。

ポインタ同士の足し算はできませんが、引き算はその二つのポインタが示す番地間にある要素数を表します。例えば

```
        int a[5];
        int* p1 = &a[1];
        int* p2 = &a[3];
        int n = p2 - p1;
```

プログラム 4.3: ポインタによる配列要素の表示

```
1 #include <iostream>
2 int main()
3 {
4         int a[5] = { 5, 4, 3, 2, 1};
5         int* p = a;
6         for (int i = 0; i < 5; i++)
7                 std::cout << "a[" << i << "] = " << *p++ << std::endl;
8         return 0;
9 }
```

プログラム 4.4: ポインタの差の利用例

```
1 #include <iostream>
2 int main()
3 {
4         int a[5] = { 5, 4, 3, 2, 1};
5         for (int* p = a; (p - a) < 5; p++)
6                 std::cout << "a[" << (p - a) << "] = " << *p << std::endl;
7         return 0;
8 }
```

とした時、n の値は 2 になります。

4.3.3 配列の番地

配列を例えば

```
        int a[5];
```

として確保した時、識別名 a 自身は、配列の格納域の番地（つまり &a[0]）を表しています。従って、

```
        int* p = &a[0];
```

と書く代わりに

```
        int* p = a;
```

と書くことができます。更に a の内容は番地ですので、a[3] は *(a + 3) と書くこともできます。逆に、p がポインタ型変数であるとき、*(p + 2) は p[2] と書くこともできます。

このようにポインタと配列は非常に強い関係がありますが、a はあくまでも配列を指し示すオブジェクトであって、値を変えるような操作はできません。一方、ポインタの方は値を変えることができます。プログラム 4.3 は プログラム 4.2 をポインタを使って書き換えた例です。この例では p に対し、単項演算子の * と ++ を使って*p++ としています。後置の ++ は *p が実行された後実行されることに注意します。

ポインタの差はそれらの番地間にある要素数を表すことを利用してプログラム 4.3 は更にプログラム 4.4 のように書き換えることもできます。

4.4 多次元配列

C++ では例えば 2 次元配列は配列の配列という考え方をします。

```
int a[3][5];
```

は整数型オブジェクトを 5 個格納する配列型オブジェクトを 3 個配列として並べたオブジェクトです。添え字と番地の対応は、一番右側の [] が先に変化します。つまり上の例を番地順で言うと、a[0][0]、a[0][1]、...、a[0][4]、a[1][0]、 ... と並んでいます。

更に次元を増やす場合は、右側に [] 内に要素数を示して加えていきます。

4.4.1 多次元配列の初期化

多次元配列の初期化においても配列の配列という考え方をします。

```
int a[3][5] =
        {
                { 5, 4, 3, 2, 1},
                { 1, 2, 3, 4, 5},
                { 3, 2, 1, 5, 4}
        };
```

は整数 5 個の配列 3 個を配列にしたものの初期化の例です。

4.4.2 ポインタの配列

配列はポインタと強い関係があることは前に述べました。その議論を踏襲すると、

```
int a[3][5];
int* p[3];
p[0] = &a[0][0];
p[1] = &a[1][0];
p[2] = &a[2][0];
```

となります。また、&a[0][0] は a[0] と同じ意味ですから、

```
int a[3][5];
int* p[3];
p[0] = a[0];
p[1] = a[1];
p[2] = a[2];
```

と書くこともできます。ここで、p の各要素が指し示す先は整数型へのポインタであって、各々独立であってもよいことに注意します。例えば 3 つの独立した配列に対し、

```
int a[3];
int b[7];
int c[5];
int* p[3];
p[0] = a;
p[1] = b;
p[2] = c;
```

のようにすることも可能です。

プログラム 4.5: ポインタ引数

```
1  #include <iostream>
2
3  int func1(int x)
4  {
5          x = x + 5;
6          return x;
7  }
8
9  int func2(int * x)
10 {
11         *x = *x + 5;
12         return *x;
13 }
14
15 int main()
16 {
17         int rval;
18         int a = 2;
19
20         rval = func1(a);
21         std::cout << "return value = " << rval << " a = " << a << std::endl;
22
23         rval = func2(&a);
24         std::cout << "return value = " << rval << " a = " << a << std::endl;
25
26         return 0;
27 }
```

4.5 関数の引数とポインタ

C/C++ では関数の引数は値渡しです。関数を呼び出す前に、引数分の一時記憶域を確保し、引数リストに書かれた引数の値をコピーしてから関数を呼び出します。この時、関数に渡すのは値をコピーした一時記憶域です。従って、関数の中で引数の値を変更しても呼び出し側の値は変わりません。しかし、引数としてポインタが渡されると、そのポインタの指す先の値を変更することが可能になります。つまり、関数の中からその関数の外にある変数値を変えることが可能になります。

プログラム 4.5 には、二つの関数 `func1` と `func2` があります。前者の引数は整数の値渡が渡りますが、後者は整数へのポインタが渡ります。関数 `func2` では、ポインタの指し示す先の値を変更しています。従って、この関数を呼び出した後では、`a` の値が変わります。

4.6 参照

C++ では、参照（reference）という型があります。

```
        int a;
```

と書くと、記憶域に整数型オブジェクトを格納する記憶域が割り当てられ、以後その記憶域の値は識別子 `a` で読み書きできることはこれまで述べてきました。いま例えば `a` に割り当てられた記憶域が 102 番地だとすると、`a` の意味は「102 番地に格納されている整数値」という意味になります。

これに対し、

プログラム 4.6: 参照の簡単な例

```
1 #include <iostream>
2 int main()
3 {
4         int a = 5;
5         int & p = a;
6         p = 3;
7         std::cout << a << std::endl;
8         return 0;
9 }
```

```
        int a;
        int & p = a;
```

と書くと、p も a と同じ記憶域を指すようになります。a に割り当てられた記憶域が 102 番地のとき、p の意味も「102 番地に格納されている整数値」という意味になります。このとき、p は a の参照であると言います。また p は a の別名（alias）であるという言い方をすることもあります。

参照は同じ記憶域を指すため値の変更は互いに影響を受けます。例を見てみましょう。プログラム 4.6 では、6 行目で p に 3 を代入していますが、p は a の参照であるので、a の値も 3 になります。

4.6.1 関数の引数と参照

関数の引数を参照とすることができます。この時、関数の引数を格納する一時記憶域には、値のコピーではなくアドレスのコピーが格納されるので、複雑な構造を持つオブジェクトの場合などでは高速化される可能性があります。

プログラム 4.7: 文字配列の初期化と要素数の表示

```
1  #include <iostream>
2  int main()
3  {
4          char s1[] = {'H','e','l','l','o'};
5          char s2[] = {"Hello"};
6          char s3[] = "Hello";
7          std::cout
8                  << "sizeof(s1) = " << sizeof(s1) << '\n';
9                  << "sizeof(s2) = " << sizeof(s2) << '\n';
10                 << "sizeof(s3) = " << sizeof(s3) << std::endl;
11         return 0;
12 }
```

4.7 文字配列と文字列リテラル

文字配列と文字列リテラルには密接な関係があります。文字配列の初期化は整数などと同じように、一文字ずつ区切って初期化することもできますが、文字列リテラルによる初期化もできます。

プログラム 4.7 では 3 つの文字配列を初期化し、その sizeof の値、すなわち（char 型の sizeof は定義により 1 なので）文字要素の個数を調べています。一見同じ文字配列のように見えますが、s1 は 5 であるのに対し、s1 と s2 は 6 となります。どうやら、文字配列というだけでは文字列というわけにはいかないようです。

この違いを見るために、文字配列の要素を整数で表示するプログラムを作ってみます。プログラム 4.8 では 3 つの文字配列を表示したいので、繰り返し同じことを書くのを避けるために関数 dumpstr を用意して、呼び出すようにしてあります。実行結果は、筆者の場合では

```
C:\cppintro>a
s1[] = 72,101,108,108,111,
s2[] = 72,101,108,108,111,0,
s3[] = 72,101,108,108,111,0,

C:\cppintro>
```

のようになりました。つまり文字列リテラルで初期化した文字配列の場合は最後に数値の 0 が入っています。

プログラム 4.8: 文字配列の要素の整数表示

```
1  #include <iostream>
2  void dumpstr(char s[], int n);
3  int main()
4  {
5          char s1[] = { 'H', 'e', 'l', 'l', 'o'};
6          char s2[] = { "Hello" };
7          char s3[] = "Hello";
8          std::cout << "s1[] = "; dumpstr( s1, sizeof(s1) );
9          std::cout << "s2[] = "; dumpstr( s2, sizeof(s2) );
10         std::cout << "s3[] = "; dumpstr( s3, sizeof(s3) );
11         return 0;
12 }
13
14 void dumpstr(char s[], int n)
15 {
16         for(int i = 0; i < n; i++)
17         {
18                 int c = s[i];
19                 std::cout << c << ',';
20         }
21         std::cout << std::endl;
22         return;
23 }
```

第5章　実行の流れの制御

C++ プログラムは基本的には書かれている順に実行されます。これだけでは単純な処理しかできませんので、実行の流れを条件に応じて選択するための文や繰り返しを行なうための文が用意されています。

5.1　選択

特定の条件を満たした時に実行の流れを選択するための文として C++ では

- `if` 文
- `switch` 文

二つが用意されています。

前者は、条件を `bool` 値で評価しその結果に応じて流れを二択的に選択ためのものです。条件を `bool` 値で評価するので、基本的には二分岐になります。

後者は、整数型もしくは列挙型の値に応じて実行の流れを選択するもので、通常は多分岐に用いられます。

ただし二分岐は多分岐の一部ですし、一方、二分岐の組み合わせで多分岐を実現できるので、どちらを使わなければならないということはありません。

5.1.1　二分岐

二分岐に通常使われるのは `if` 文です。これは

```
if (条件)
        文1
```

という構成、または

```
if (条件)
        文1
else
        文2
```

という構成をとります。いずれも "条件" の部分を `bool` 値に変換して、その値が `true` の時のみ、"文 1"を実行します。そうでない時、すなわち "条件" の部分を `bool` 値に変換した時、その値が `false` の時は、"文 1"は実行されずに、前者の場合は次の文へ制御が移ります。また後者の場合は"文 2"が実行された後、次の文へ制御が移ります。

これらは、いずれも全体で単一の文として扱われることに注意します。このことから、条件の入れ子構造を単一の文として作ることができます。例えば

```
if (条件1)
        if (条件2)
                文1
        else
                文2
```

も単一の文です。このような構文の場合、else はそれより前にある if のうち、最も近い if と結びつけられます。この例の場合の else は ”条件 2” の if に対する else です。つまり、”文 2” が実行されるのは、”条件 1” が true であり、かつ ”条件 2” が false である時です。

改行や字下げは単に読みやすくしているだけであることに注意します。これを間違えると、意味の読み間違えを非常に犯しやすくなります。例えば今の例を

```
if (条件1)
        if (条件2)
                文1
else
        文2
```

と書いても、文法的に全く同じですが、見た目には ”文 2” の前の else は、”条件 1” の if と結びついているような印象を受け ”文 2” は ”条件 1” が false であれば、”条件 2” に係わりなく実行されると誤解しやすくなります。

他に非常によく使われる構文として

```
if (条件1)
        文1
else if (条件2)
        文2
else if (条件3)
        文3
        :
else
        文4
```

があります。この例の場合も、else は最も直前にある if と結び付けられるという規則で読み取ります。例えば”文 3” が実行されるのは、”条件 1” が false で、”条件 2” も false で、”条件 3” が true の時であることがわかります。

5.1.2 多分岐

整数型または列挙型の値に応じて、実行を変えたい場合、if-else 構文を使って書くこともできますが、より単純に（従って、より読みやすく）書く手段が提供されています。それが switch 文です。

```
switch (条件) 文
```

という形をとり、これ全体で一つの文とみなされます。従って if 文と同様、入れ子構造をとることができます。

条件は評価された後、値が保持され、それに続く文の中にある case ラベルと比較され、値が一致するラベルに続く文が実行されます。一致する値が無い場合は default ラベルに続く文が実行されます。通常よく使われる構文は

```
switch (条件)
{
case 値1:
        文1
case 値2:
        文2
case 値3:
        文3
        :
default:
        文4
}
```

です。条件が評価され、その値が ”値 2” だとすると、”文 2”、”文 3”、...、”文 4” と実行されます。”文 2” で switch 文から抜けるわけではありません（fall through 方式と言います)。つまり case ラベルはあくまでも実行の開始位置を示しているだけで、本体部分には何ら影響を与えません。このことから、switch 文の本体でオブジェクトを定義する場合には細心の注意が必要です。なぜなら、条件によってそのコンストラクタが実行されない可能性があるからです。例えば上の例で、”文 1”でオブジェクトを定義し、”文 2”でもそれを使っている場合、条件を評価して”値 2” が得られたとすると、そのオブジェクトのコンストラクタは”文 1”にあるので、実行されない可能性があります（結果は不定)。基本的には switch 文の本体でオブジェクトを定義してはいけません。

途中で switch 文から抜けるには break 文を使います。例えば上の例で、各 case の文を実行したら他の case には入らずにただちに switch 文を抜けるようにするには

```
switch (条件)
{
case 値1:
        文1
        break;
case 値2:
        文2
        break;
case 値3:
        文3
        break;
        :
default:
        文4
        break;
}
```

のように書きます。ここで、最後のラベル（今の場合 defalut ラベル）での break は不要ですが、順番を入れ替えたり、ラベルを追加したときなどに忘れがちなので、最後であっても入れておくのがよいでしょう。

なお、ラベルはいくつあっても構いませんし、無くても構いませんが、一つの値に対しては最大でも一つである必要があります。同じ値の case ラベルが二つ以上あってはいけません。また default ラベルが複数あってもいけません。条件を評価した時、一致する値の case ラベルが無く、かつ default ラベルも無い場合は、何も実行せずに、switch 文の直後の文に制御が移ります。

5.2 繰り返し

ある条件が満たされている間、あるいはある条件が満たされるまで繰り返し実行したいという場合があります。次の三つの文は、これを実現するものです。

- for 文
- while 文
- do 文

このうち、for 文と while 文は、各繰り返しの開始前に繰り返しの条件を評価します。一方 do 文は、各繰り返しの最後で繰り返しの条件を評価するので、最低でも一回は do 文に含まれる文を実行します。

繰り返しの実行を途中で制御するために

- break 文
- continue 文

があります。このうち break 文は繰り返し文を終了し、その直後の文に制御を移します。一方 continue 文は、繰り返しの残り部分をスキップして繰り返し条件の再評価に進み、条件が満たされていれば繰り返します。

繰り返し文は、論理的にはどれを使っても同じことが実現できますが、繰り返しの型により表現しやすいものとしにくいものがあります。以下では、これらを念頭に置いて、この三つの文の書き方を見てみましょう。

5.2.1 カウント型の繰り返し

カウント型の繰り返しというのは、繰り返し回数を指定して繰り返すような場合です。これを一般化したものが for 文です。

```
for (初期化部; 条件部; 反復式) 文
```

という構文をとります。まず初期化部が実行され、続いて条件部の評価が行われます。評価の結果が true であれば ”文” を実行し、続いて反復式を評価し、条件部の評価に戻ります。こうして条件部の評価が false になるまで繰り返されます。

次の例は 100 個の整数配列 ia 順に整数値 1 から 100 までを格納する例です。

```
int ia[100];
for (int i = 0; i < 100; i++)
        ia[i] = i + 1;
```

ここに示した例のように、初期化部には文だけでなく宣言も含めることができます。初期化部で宣言された変数名の有効範囲はこの for 文の文末までです。

初期化部、条件部、反復式はいずれも省略可能です。条件部が省略された場合は、true であると見なされます。つまり無限ループになります。例えば

```
for (;;)
```

は無限ループを表します。この場合、繰り返し本文中のどこかに break が無いと本当に無限ループ（止まらないプログラム）になります。

繰り返しの本文中で break が実行された場合、その時点で for 文を終了し、この for 文の直後にある文に制御を移します。この時、反復式の部分は評価されません。

繰り返しの本文中で continue が実行された場合、繰り返し本文の残りの部分をスキップし、反復式を評価し、続いて条件部を評価します。条件部が true であれば繰り返し本文の先頭から実行し、false であれば、for 文を終了し、この for 文の直後にある文に制御を移します。

5.2.2 繰り返し回数が事前にわからない場合の繰り返し

繰り返し回数が事前にはわからない場合は通常 while 文が使われます。これは

while（条件部） 文

という構文をとります。まず条件部が評価され、その値が true であれば繰り返し本文の ”文” が実行され、条件部が再評価されます。もし false であれば ”文” は実行されずに while 文を終了し、この while 文の直後にある文に制御を移します。

繰り返し回数が事前にわからない場合であっても、for 文と break 文を組み合わせることで対応はできますが、このような場合は通常 while 文の方がわかりやすく書けます。

繰り返し本文中で break 文が実行された場合、その時点で while 文を終了し、この while 文の直後にある文に制御を移します。また continue 文が実行された場合、繰り返し本文の残りの部分はスキップされ、条件部が再評価されます。もし評価の結果が true であれば繰り返し本文の先頭から再実行されます。評価の結果が false であれば while 文を終了し、この while 文の直後にある文に制御を移します。

5.2.3 事後評価による繰り返し

繰り返しの中には、何かを実行して、その結果によって更に繰り返すかどうかを決める場合があります。このような場合は、事後に繰り返し条件を評価する do 文が使われます。この繰り返し文は

do 文 **while**（条件式）

という構文をとります。繰り返しの本文である ”文” は少なくとも 1 回実行され、その後条件式が評価されます。評価の結果 true であれば繰り返し本文である ”文” が再度実行され、false であれば、do 文を終了して、この do 文の直後にある文に制御を移します。

繰り返し本文中での break 文、continue 文の実行による振る舞いは、while 文の場合と同様です。すなわち、繰り返し本文中で break 文が実行された場合、その時点で do 文を終了し、この do 文の直後にある文に制御を移します。また continue 文が実行された場合、繰り返し本文の残りの部分はスキップされ、条件式が再評価されます。もし評価の結果が true であれば繰り返し本文の先頭から再実行されます。評価の結果が false であれば do 文を終了し、この do 文の直後にある文に制御を移します。

5.3 飛び越し

制御を一気に別の箇所に飛び越すための文としては、すでに述べた

- continue 文
- break 文
- return 文

がありますが、この他に同じ関数内にある別の箇所に飛び越すための文として goto 文があります。

```
goto 識別子
```

という構文をとります。ここで識別子はラベルにより指定されるものです。ラベルによる指定は

```
識別子: 文
```

という構文になります。識別子が一致した場合、goto 文からこのラベルの文へ制御が一気に移動します。この時、オブジェクトの定義を飛び越えてしまう可能性があることに注意します。つまりコンストラクタが呼び出されない可能性があります。このようなオブジェクトを使用した場合どのような振る舞いになるかは不定です。このような危険性がありますので、goto 文の使用は避けた方が無難です。

通常 goto 文が使われるのは、深い入れ子の中から一気に抜けたい場合です。例えば以下の例は 16x8 の整数の二次元配列のデータをひとつずつ標準入力から読み取るプログラムで途中で読み取りエラーがあったら一気に繰り返し制御から抜けます。

```
#include <iostream>
int main()
{
        int grayimage[16][8];
        for (int h = 0; h < 16 ; h++)
                for (int w = 0; w < 8; w++)
                        if (!(std::cin >> grayimage[h][w]))
                                goto error;
        goto alldone;
error:
        std::cerr << "Short data. Needs 128 integers." << std::endl;
alldone:
        return 0;
}
```

しかし、このような場合であっても、後述する例外機構を使えば goto 文より安全な方法で実現できます。

第6章　クラス

実際の処理を行う時に処理の対象が単純な一つの型の数値データだけで構成されることは稀です。例えば観測データの場合でも、測定値の他に時刻情報や測定条件などを含む場合があるでしょう。またデータだけでなく、データに対する固有の処理も存在するでしょう。

ある性質を表すデータの組を持っていて、それらに対する処理を一つにまとめて表したものをC++ ではクラスと呼びます。クラスがどのようなものかもう少し詳しく見るために、簡単な例題を通して説明していきます。

6.1　単純なクラスを作ってみる

制御や計測をやっていると、値に上限値と下限値が存在することがよくあります。まずこれを一つのクラスにしてみましょう。プログラム 6.1 を見てください。3 行目から 8 行目で、Range という名前のクラスを定義しています。

6.1.1　クラス名の導入

最初の

```
class Range
```

は、これから導入するクラスの名前を与えています。このクラスの名前 Range は、筆者が勝手に決めたもので、名前付けの規則さえ守っていれば自由につけることができます。アルファベットの大文字と小文字は区別されることに注意してください。ここではクラス名の最初の文字を大文字にしていますが、こういう規則があるわけではありません。ただしクラス名と変数名は区別する必要があります。そこで、ここでのルールとして、変数名は全部小文字、クラス名の方は少なくとも先頭 1 文字は大文字にしておこうというぐらいの意味です。

6.1.2　メンバ変数

プログラム 6.1 の 6 行目と 7 行目は、このクラスには二つの double 型の変数 m_lower と m_upper を持っていることを示しています。このようにクラスが保持する変数をメンバ変数と言います。その前にある

```
public:
```

は、アクセス指定子と呼ばれるもので、これ以降のメンバ変数は公開（public）であることを宣言しています。公開というのは、このクラスの外から何ら制約なくアクセスできるということです。このことについては次の節で述べます。

プログラム 6.1: 単純な Range class

```
1  #include <iostream>
2
3  class Range
4  {
5  public:
6          double m_lower;
7          double m_upper;
8  };
9
10 int main()
11 {
12         Range range1;
13
14         range1.m_lower = 10.0;
15         range1.m_upper = 90.0;
16
17         Range range2 = range1;
18
19         std::cout << "lower=" << range2.m_lower << std::endl;
20         std::cout << "upper=" << range2.m_upper << std::endl;
21
22         return 0;
23 }
```

このアクセス指定子は次のアクセス指定子が現れるまで有効です。つまり今の場合、メンバ変数 m_lower も m_upper もどちらも公開メンバ変数です。

メンバ変数の名前の先頭を m_ で始めてますが、メンバ変数であることを区別しやすいように筆者が勝手につけているだけで、規則ではありません。しかし、一般によく使われているやり方です。

6.1.3 メンバ変数へのアクセス

さて、以上で新たなデータ型である Range クラス型が導入できました。そこで、使用例を見てみましょう。

プログラム 6.1 の 10 行目以降が使用例になります。12 行目で、range1 という Range 型の変数を宣言していています。これは整数型の変数などを宣言するのと同じ書き方です。14 行目と 15 行目で、このオブジェクトのメンバ変数に値を代入しています。このように、メンバ変数を指定するにはオブジェクトの名前の後にドットを付けてメンバ変数の名前を書きます。

17 行目は、新たな Range 型の変数 range2 を定義して、その値を range1 で初期化しています。一つの代入文で全部のメンバ変数の値が代入されます。

19 行目と 20 行目では、代入が正しく行われたことを見るために、vol2 のメンバ変数を一つずつ標準出力へ出力しています。メンバ変数を指定するのにオブジェクトの名前の後にドットを付け、その後にメンバ変数名を書くのは先の場合と同じです。

さて、ここで、main 関数の中で、Range 型オブジェクトのメンバ変数名を直接書くことでアクセスできるのはプログラム 6.1 の 10 行目で、これらが public と指定されているからです。もし 10 行目が無いと、これらの変数は private とみなされ、クラスの外からはアクセスできなくなります。つまり、14 行目、15 行目、19 行目、20 行目はエラーになります。

6.1.4 メンバ関数

クラスにまとめあげることができるのは、データだけではありません。そのクラスに固有の関数を含ませることができます。この関数をメンバ関数と呼びます。ここでは今作った Range クラスに、ある値がその最大値と最小値の範囲にある時、bool 値 true を返し、そうでない時には false を返す関数 is_inside を導入しましょう。

プログラム 6.2 には、メンバ関数 is_inside を導入し、その使用例も合わせて示してあります。メンバ関数の定義自身は、クラスの定義とは別に与えていることに注意します。この時、メンバ関数の定義を与える関数名はクラス名を先頭につけ、その後に :: を付け、関数名を指定します。関数の引数リストの後に const と付いていますが、これはこの関数がオブジェクトの値を変えない、つまりどのメンバ変数の値も変えないということを意味します。

さて、このようにクラスのメンバ関数にしなかった場合を考えてみましょう。その場合は、関数とデータが結びついていないので、必要なデータは関数の外から与えるしかありません。例えば

```
bool is_inside(double x, double xmin, double xmax);
```

のような関数を作ることになるでしょう。つまり、この関数を呼び出す度に判定に必要となる上限値、下限値を与える必要があります。もし多数の Range が必要な場合は、呼び出す度に正しい対象の上限値、下限値を与える必要があります。

一方、クラスのメンバ関数にした場合、上限値、下限値は一度設定した後は気にする必要はありません。ある値が対象となる範囲に入っているか否かは対象とする範囲の名前とその値だけで決まります。対象とする範囲の上限値、下限値をいくつに設定したかを思い出しす必要はありません。必要最小限の情報だけに専念できるわけです。これにより、プログラムを書く方も読む方も見通しがずっとよくなります。

6.2 コンストラクタ

前節で作った Range クラスは上限値と下限値を保持するものです。この値は、公開されていて、このクラスの外から簡単に変更できてしまいます。これでは困ることもあるでしょう。ここでは、一度設定したら変更できないようにしてみましょう。

クラスのメンバを外から見えないようにするには、前節で使ったアクセス指定子 public の代わりに private を使います。

```
class Range
{
private:
        double m_lower;
        double m_upper;
public:
        bool is_inside(double x) const;
};
```

このように書くと、メンバ関数 is_inside だけがクラスの外から使うことができて、メンバ変数の m_lower と m_upper はクラスの外からは読み書きできなくなります。この場合でも、is_inside は、このクラスのメンバ関数ですので、この関数の中では使うことができます。従って、is_inside の定義は変更する必要はありません。

プログラム 6.2: メンバ関数の導入

```
1  #include <iostream>
2
3  class Range
4  {
5  public:
6          double m_lower;
7          double m_upper;
8          bool is_inside(double x) const;
9  };
10
11 bool Range::is_inside(double x) const
12 {
13         if((x < m_lower) || (x > m_upper))
14                 return false;
15         return true;
16 }
17
18 int main()
19 {
20         Range range;
21
22         range.m_lower = 10.0;
23         range.m_upper = 90.0;
24
25         std::cout << "lower=" << range.m_lower << std::endl;
26         std::cout << "upper=" << range.m_upper << std::endl;
27
28         double x;
29         while((std::cin >> x))
30         {
31                 std::cout << "value(" << x << ")␣is␣";
32                 if(range.is_inside(x))
33                         std::cout << "inside";
34                 else
35                         std::cout << "outside";
36                 std::cout << std::endl;
37         }
38         return 0;
39 }
```

さてこれでメンバ変数の m_lower と m_upper は変更できなくなりましたが、これでは設定もできません。これを解決する手段としてコンストラクタ (constructor) があります。コンストラクタは、そのクラスのオブジェクトが生成される時にメンバの初期化を行う手段を提供します。プログラム 6.3 はコンストラクタの導入とその使用例を示しています。

コンストラクタはクラス名と同じ名前のメンバ関数として記述します。ただし戻り値は指定できません（void 型を指定することもできません)。プログラム 6.3 では、6 行目の

```
        Range(double upper, double lower);
};
```

で double 型の引数を二つ持つコンストラクタを宣言し、13 行目から 16 行目の

```
Range::Range(double lower, double upper) :
        m_lower(lower), m_upper(upper)
{
}
```

で、その定義を与えてます。ここで関数の引数リストの後、コロン : に続けて double 型のメンバー変数を引数を使って初期化しています。メンバのコンストラクタは、それらを含むクラスのコンストラクタが実行されるより前に呼び出されます。ここで示した記法を使うことにより、メンバの生成時点で初期値を与えることができます。

```
Range::Range(double lower, double upper)
{
        m_lower = lower;
        m_upper = upper;
}
```

と書いても文法的には問題ありませんが、この場合は m_lower と m_upper が（既定値で）生成された後、Range の生成の途中で値が**代入**されます。

このクラスのオブジェクトを定義するには引数を二つ与える必要があります。プログラム 6.3 の 27 行目で

```
        Range vol(10.0, 90.0);
```

として、Range クラスの vol という名前のオブジェクトを生成しています。この結果、下限値 10.0、上限値 90.0 を持つ vol という名前の Range 型オブジェクトが生成されます。

6.2.1 引数無しのコンストラクタ

さて、Range 型の配列を作ってみます。例えば 3 つの Range 型要素を持つ配列を作るため

```
        Range rarray[3];
```

とすると、コンパイルエラーになります。これは引数無しで Range 型オブジェクトを作ろうとするためです。

```
        Range rarray[3] = {Ranage(10.0, 90.0), Range(20.0, 80.0), Range(30.0, 70.0)};
```

のように、引数付きのコンストラクタで初期化すれば作ることができます。しかし、配列だけ確保しておいて、後から値を代入したいこともあるでしょう。このような時には

プログラム 6.3: コンストラクタの導入

```
1  #include <iostream>
2
3  class Range
4  {
5  public:
6          Range(double lower, double upper);
7          bool is_inside(double x) const;
8  private:
9          double m_lower;
10         double m_upper;
11 };
12
13 Range::Range(double lower, double upper) :
14         m_lower(lower), m_upper(upper)
15 {
16 }
17
18 bool Range::is_inside(double x) const
19 {
20         if((x < m_lower) || (x > m_upper))
21                 return false;
22         return true;
23 }
24
25 int main()
26 {
27         Range range(10.0, 90.0);
28
29         double x;
30         while((std::cin >> x))
31         {
32                 std::cout << "value(" << x << ") is ";
33                 if(range.is_inside(x))
34                         std::cout << "inside";
35                 else
36                         std::cout << "outside";
37                 std::cout << std::endl;
38         }
39         return 0;
40 }
```

```
        Range rarray[3];
        rarray[0] = Range(10.0,90.0);
        rarray[1] = Range(20.0,80.0);
        rarray[2] = Range(30.0,70.0);
```

のように書けたら便利です。

```
        Range range;
```

のように、引数無しで定義されるクラスオブジェクトの場合は、引数無しのコンストラクタが呼ばれます。実は、このコンストラクタは、クラス定義にコンストラクタの定義が一つも含まれていない場合には処理系により自動的に生成されますが、一つでもコンストラクタが定義されていると、自動的には生成されません。従って、引数付きのコンストラクタも引数無しのコンストラクタも使いたい場合は、両方定義する必要があります。

6.3 インラインメンバ関数

前節までで、Range クラスのメンバ変数は、オブジェクト生成時に一度設定したら、後は変更できなくなりました。それはそれで目的を果たしたのでよいのですが、クラスの外からは下限値、上限値を知ることすらできなくなりました。そこで、これらを関数を通じて読めるようにしましょう。

プログラム 6.4 では、lower と upper というメンバ関数を用意しました。これらはそれぞれ単にメンバ変数 m_lower と m_upper の値を返すだけですが、 is_inside と違って、クラス定義の中に本体まで記載されています。このようにクラス定義の中に関数の本体まで与えると、コンパイラはインラインメンバ関数として、インライン展開を試みます（必ずそうするというわけではありません)。インライン展開というのは、今の例の場合

```
        Range vol(10.0, 90.0);
        double xmin = vol.lower();
```

とあった時、

```
        Range vol(10.0, 90.0);
        double xmin = vol.m_lower;
```

と同等の翻訳を行うということです。つまり、m_lower は private なので、vol.m_lower とは書けませんが、それと同程度のことをメンバ関数により実現できます。もちろんインラインメンバ関数とせずに is_inside と同様に、クラス定義の中では

```
        double lower() const;
        double upper() const;
```

としておいて、関数の本体は別に

```
double Range::lower() const
{
        return m_lower;
}
double Range::upper() const
{
        return m_upper;
}
```

プログラム 6.4: インラインメンバ関数の導入

```
1  #include <iostream>
2
3  class Range
4  {
5  public:
6          Range(double lower, double upper);
7          double lower() const { return m_lower; };
8          double upper() const { return m_upper; };
9          bool is_inside(double x) const;
10 private:
11         double m_lower;
12         double m_upper;
13 };
14
15 Range::Range(double lower, double upper)
16         : m_lower(lower), m_upper(upper)
17 {
18 }
19
20 bool Range::is_inside(double x) const
21 {
22         if((x < m_lower) || (x > m_upper))
23                 return false;
24         return true;
25 }
26
27 int main()
28 {
29         Range range(10.0, 90.0);
30
31         std::cout << "lower=" << range.lower() << std::endl;
32         std::cout << "upper=" << range.upper() << std::endl;
33
34         double x;
35         while((std::cin >> x))
36         {
37                 std::cout << "value(" << x << ")␣is␣";
38                 if(range.is_inside(x))
39                         std::cout << "inside";
40                 else
41                         std::cout << "outside";
42                 std::cout << std::endl;
43         }
44         return 0;
45 }
```

と与えることもできます。しかし、この場合は、あくまでも関数呼び出しです。インライン関数は（インライン展開するので）その定義は各翻訳単位に含まれている必要があります（そのため通常はヘッダに書きます）が、インラインで無い関数の定義はプログラム全体で一つでなければなりません。

6.4 構造体とクラス

クラスの同義語で構造体と呼ばれているものがあります。違いは、メンバに対するアクセス指定子を省略した場合、クラスでは `private` であるのに対し、構造体では `public` であるという点だけです。例を示します。

```
struct Point
{
        int m_x;
        int m_y;
};
```

は、

```
class Point
{
public:
        int m_x;
        int m_y;
};
```

と文法上全く同じ意味です。

ではなぜ同じものを表現するのに、このような二つの書き方があるのかというと、C 言語のソースコードがそのまま使えるようにするためです。C 言語で使われていた構造体は、そのまま C++ の構造体として使うことができます。ただし逆は成り立ちません。C 言語にはクラスの概念はありませんので、C++ で書かれた構造体は、必ずしも C 言語の構造体としては使えません。例えば C++ の構造体はクラスの別の書き方というだけですので、コンストラクタやデストラクタをはじめ、メンバー関数を持つこともできますし、アクセス指定子を使ってメンバーにアクセス制御を行うこともできます。継承や派生もできます。このような使われ方をした C++ の構造体は、もはや単純に C 言語の構造体として使うことはできなくなります。

第7章　名前空間と分割コンパイル

作成するプログラムの規模が大きくなったり、分担してプログラムを開発する場合、プログラムの一部をコンパイルしてテストする必要が生じます。また、識別子などの衝突を防ぐ必要も生じます。ここでは、その方法について述べます。

7.1　分割コンパイル

分割コンパイルというのは、ソースファイルを複数のファイルに分けて、別々にコンパイルすることを言います。特にクラスなどを定義した場合、クラスや関連するクラスごとに一つのファイルにする方が管理も楽になり、分担してプログラム開発を行うことも可能になります。

ソースファイルを分けると言っても、言語系が翻訳できる単位で分けなければなりません。また、例えば変数の型などは翻訳単位ごとに宣言されていないと、翻訳できません。ここでは、いままで作ってきた Range クラスを例にとって、説明していきます。

7.1.1　ヘッダファイルを作る

クラスの宣言などは、それが使われる翻訳単位ごとに行う必要があります。このような部分は、通常はヘッダファイルとして別ファイルにし、#include を使って翻訳単位ごとに読み込むようにします。

例えば、我々の Range クラスの場合、

```
class Range
{
public:
        Range();
        Range(double lower, double upper);
        double & lower() { return m_lower; };
        double & upper() { return m_upper; };
        bool is_inside(double x) const;
private:
        double m_lower;
        double m_upper;
};
```

の部分をヘッダファイルとして切り離すのがよいでしょう。ところが、このままだと少し問題があります。プログラムが大規模になってくるにつれ、あるクラスを利用した別のクラスを作ったり、それらを集めて、また一つの翻訳単位を構成するようになります。このような場合、ヘッダファイルの中に更に別のヘッダファイルを必要とすることがあります。すると、同じヘッダファイルを二重、三重に読み込むことになることがあります。同じ宣言が何度も繰り返されるのは無駄なだけでエラーにはなりませんが、多重にインクルードされたファイルに一つでも定義が含まれていると、

多重定義のエラーになって翻訳できません。そこで、通常は、#ifdef と #endif マクロを使ってこれを防ぎます。これを入れたものをプログラム 7.1 に示します。

プログラム 7.1: Range クラスのヘッダファイル

```
1  #ifndef RANGE_H_INCLUDED
2  #define RANGE_H_INCLUDED
3
4  class Range
5  {
6  public:
7          Range();
8          Range(double lower, double upper);
9          double lower() const { return m_lower; };
10         double upper() const { return m_upper; };
11         bool is_inside(double x) const;
12 private:
13         double m_lower;
14         double m_upper;
15 };
16
17 #endif  // RANGE_H_INCLUDED
```

この 1 行目の意味は、もし RANGE_H_INCLUDED という変数（翻訳の際に処理系が使う変数で、プログラム中での変数ではありません）が定義されていなければ以下を翻訳し、そうでなければ #endif までスキップしなさいという意味です。次の行では（2 行目）#define を使って、RANGE_H_INCLUDED を定義しています。つまり、このファイルが多重にインクルードされていても、2 回目以降は全部スキップされ多重インクルードに起因するエラーを防ぐことができます。

7.1.2 ソースファイルの分離

さて、ヘッダを分離したので、残りは比較的簡単です。我々の Range クラスの各メンバー関数を定義している部分と、Range クラスを使う部分（いまの場合は main 関数のみ）とに分けるのが合理的です。この時、各々のファイルにプログラム 7.1 を include します。

プログラム 7.2: Range クラスのヘッダファイル

```
1  #include "range.h"
2
3  Range::Range() :
4          m_lower(0.0), m_upper(0.0)
5  {
6  }
7
8  Range::Range(double lower, double upper) :
9          m_lower(lower), m_upper(upper)
10 {
11 }
12
13 bool Range::is_inside(double x) const
14 {
15         if((x < m_lower) || (x > m_upper))
16                 return false;
17         return true;
18 }
```

プログラム 7.3: Range クラスを使う

```
1  #include <iostream>
2  #include "range.h"
3
4  int main()
5  {
6          Range range[3];
7
8          range[0] = Range(10.0,90.0);
9          range[1] = Range(20.0,80.0);
10         range[2] = Range(30.0,70.0);
11
12         for (int i = 0; i < 3; i++)
13         {
14                 std::cout << "range[" << i << "]=["
15                           << range[i].lower() << ","
16                           << range[i].upper() << ']' << std::endl;
17         }
18
19         double x;
20         while((std::cin >> x))
21         {
22                 std::cout << "value(" << x << ")␣is␣" << std::endl;
23                 for (int i = 0; i < 3; i++)
24                 {
25                         std::cout << "range[" << i << "]:";
26                         if(range[i].is_inside(x))
27                                 std::cout << "inside";
28                         else
29                                 std::cout << "outside";
30                         std::cout << std::endl;
31                 }
32         }
33         return 0;
34 }
```

プログラム 7.2 は Range クラスの各メンバー関数の定義の部分です。ここで、プログラム 7.1 は、range.h というファイル名で作成されているとしています。この時、インクルードされるファイル名の前後が、二重引用符 " で囲まれていますが、このファイルの置かれている場所が言語系が用意した場所ではなく言語系の利用者が使う場所にあることを意味しています。この書き方でのファイルの置かれている場所は、通常は現在位置ですが、他のフォルダ等に置くこともできます。ただしその場合は、処理系に依存した環境設定や、翻訳時オプションなどを指定する必要があります。

プログラム 7.3 は Range クラスを使う側のソースファイルです。ここでは簡単に 3 つの異なる Range オブジェクトを用意し、標準入力から浮動小数点数を読み取り、各々の Range オブジェクトに含まれているかいないかを判定して、その結果を表示するものです。入力に数値以外のものを入れるなど、読み取りエラーを起こさせるとプログラムを終了します。

7.1.3 コンパイルの方法

ここまでで、プログラムを 3 つのファイルに分けました。これらのファイル名を以下のようにします。

- Range クラスのヘッダファイル、プログラム 7.1 — range.h

- Range クラスの定義ファイル、プログラム 7.2 — range.cpp
- Range クラスを使うファイル、プログラム 7.3 — rangetest.cpp

ファイルは 3 つありますが、翻訳単位は range.cpp と rangetest.cpp の二つであることに注意します。range.h は #include により、これら二つに含まれています。

これら二つの翻訳単位を一度に翻訳することができます。

```
g++ rangetest.cpp range.cpp
```

これは、いままで一つのファイルで行っていたものを複数のファイル指定にしただけです。これにより、実行形式のファイル a.out（Windows の場合は a.exe ）ができます。ただし、これはファイルが分割されただけであって、分割コンパイルではありません。分割コンパイルというのは、個々の翻訳単位を個々にコンパイルすることを言います。

個々にコンパイルする場合、翻訳単位に main 関数が含まれているとは限りません。また、main 関数が含まれている翻訳単位であっても、すべての定義が含まれているわけではありません。個々にコンパイルする段階では翻訳はしますが実行形式のファイルを作るわけではありません。翻訳のみを行ったファイルを作ります。実行形式のファイルを作るには、これら個々に翻訳を行ったファイルや必要なライブラリを結びつける作業がが必要になります。これをリンクと呼ぶことがあります。また、個々にコンパイルして作成された翻訳済みのファイルをオブジェクトファイルと呼ぶことがあります。ただし、コンピュータ言語で言うオブジェクトとは全く関係ありません。

翻訳のみを行って実行形式のファイルを作らないようにするには、処理系に依存したオプションを指定します。本書で使っている g++ の場合は、-c というオプションを指定します。

```
g++ -c range.cpp
```

これにより g++ は range.o というファイルを生成します。オプションや作成されるファイルは処理系に依存しますので、他の処理系を使う場合はマニュアル等で確認してください。例えば、Microsoft の Visual Studio Express Edition 2008 に付属の C++ コンパイラでは、オプションは /c で、作成されるファイルは、今の場合 range.obj となります。

実行形式のファイルまで作るには、

```
g++ -c rangetest.cpp
g++ -c range.cpp
g++ rangetest.o range.o
```

とします。最後は、リンク作業ですが、g++ の場合は、同じ g++ で作成することができます。また、翻訳単位のいくつかだけを先にコンパイルして

```
g++ -c range.cpp
g++ rangetest.cpp range.o
```

とすることもできます。

7.2 名前空間

プログラムの規模が大きくなったり、手分けしてプログラム開発を行なう場合などでは、識別名などを管理を行なわないと名前の衝突が起こる可能性があります。このため C++ では名前の通用する範囲を設定することができます。これを名前空間（namespace ）と呼びます。

7.2.1 名前空間を定義する

名前空間を定義するには、名前空間に名前を付け、その通用範囲を {} で囲みます。例えば mylibrary という名前空間を設定するには

```
namespace mylibrary
{
int a, b;
        :
}
```

のようにします。こうすると、識別名 a や b は名前空間 mylibrary の中でしか通用しなくなります。名前空間 mylibrary の外からこれらを参照するには a や b ではなく、mylibrary::a や mylibraray::b とします。

標準出力オブジェクトは std::cout という識別名でした。これは名前空間 std に属する cout という意味です。名前空間 std は C++ の標準ライブラリが使用する名前空間の名前です。

7.2.2 名前空間の分割

同一の名前の名前空間は、分割されていても一つの名前空間を形成します。例えば前の例は

```
namespace mylibrary
{
int a;
        :
}
        :
namespace mylibrary
{
int b;
        :
}
```

としても名前空間的には同じです。また、複数の翻訳単位に分かれていても同一の名前の名前空間は一つの名前空間を形成します。

7.2.3 名前空間の入れ子

名前空間の中に更に名前空間を設定することができます。

```
namespace mylibrary
{
int a;
        namespace netlib
        {
```

```
        int c;
                :
        }
        :
}
```

などとすることができます。このとき c オブジェクトを名前空間 mylibrary の外から参照する場合は、mylibrary::netlib::c として参照します。

7.2.4 名前空間に別名をつける

名前空間には別名をつけることができます。

```
namespace mylibrary
{
int a, b;
        :
}

namespace mylib = mylibrary;
```

とすると以後 mylibrary::a の代わりに mylib::a としても参照できるようになります。これは、例えばライブラリの開発などで、バージョンを切り替えてテストする時などに便利です。ただし別名として宣言した名前で名前空間の定義を行なうことはできません。

7.2.5 名前のない名前空間

名前の無い名前空間は、翻訳単位内に限定された名前空間です。

```
#include <iostream>
namespace
{
int a = 3;
}
int main()
{
        std::cout << a << std::endl;
        return 0;
}
```

とすると、ここの a は他の翻訳単位からは見えません。翻訳単位 A にある名前の無い名前空間と、翻訳単位 B にある名前の無い名前空間とは別々の名前空間です。つまり名前の無い名前空間内の識別名は他の翻訳単位とは競合しません。

7.2.6 大域名前空間

すべての名前空間の外側で、かつすべての関数の外側で、かつすべてのクラスの外側は大域名前空間（global namespace ）と呼ばれ、すべての翻訳単位で共有されます。従って、この空間は一つのプログラムに一つだけです。大域名前空間の中の識別名であることを明示するには、単項の :: 演算子を使います。

```
#include <iostream>
namespace A
{
int a = 3;
}
int a = 5;
int main()
{
        std::cout << A::a << ',' << ::a << std::endl;
        return 0;
}
```

は、3,5 と表示されます。

7.2.7 using 宣言

using 宣言は、指定された名前を using 宣言している空間に追加します。従って、その名前は名前解決演算子:: 無しで参照できるようになります。簡単な例を示します。

```
#include <iostream>
int main()
{
        using std::cout;
        cout << "hello, world" << std::endl;
        return 0;
}
```

この例では、using std::cout; を宣言した後、cout だけで標準出力に出力しています。

7.2.8 using ディレクティブ

using ディレクティブは指定された名前空間を名前解決のための名前空間リストの最後に追加します。

```
#include <iostream>
int main()
{
        using namespace std;
        cout << "hello, world" << endl;
        return 0;
}
```

この例では名前空間 std を名前解決のための名前空間として追加しています。従って、その次の行で、通常なら std::cout や std::endl とするべきところを cout や endl で済ませることができます。

第8章　演算子の多重定義

C++ では多くの演算子はクラスオブジェクトに対しその振る舞いを定義することができます。これを演算子の多重定義、またはオーバーロード（overload）と言います。

8.1　加算の多重定義の例

我々が扱っているクラス Range に対し、加算を考えてみましょう。ただし、Range 同士の加算は常識的には和集合と思われます。そうなると二つの領域を生じることもあるので、これまで作ってきた Range クラスでは扱えません。そこでここでは Range クラスオブジェクトと浮動小数点数を加算することを考えます。この加算の意味は、領域の両端をその値だけ拡大することだとしましょう。

演算子 + は多重定義可能な二項演算子です。演算子を多重定義するには関数またはメンバ関数として定義します。この時の関数名は operator に演算記号をつけたものです。つまり今の場合、operator+ が関数名になります。まず、メンバ関数として実装してみましょう。この時左オペランドは常にそのクラスオブジェクトになります。したがって、単項演算子の場合、関数の引数はありません。二項演算子の場合は右オペランドが引数になります。いまの場合、関数の戻り値は Range クラスオブジェクトですが、（加算処理後の）自分自身の参照を返すことにします。そこで、メンバ関数としての宣言は

```
        Range& operator+(double x);
```

とします。実装は短く簡単ですのでインラインで済ませてしまいます。

```
        Range& operator+(double x)
                { m_lower -= x; m_upper += x; return *this;}
```

ヘッダのパブリックメンバー関数の宣言にこれを付け加えて完成です。

簡単なテストをしてみましょう。

```
#include <iostream>
#include "range.h"
int main()
{
        mylibrary::Range ra = mylibrary::Range(10.0, 90.0);
        std::cout << ra.lower() << ',' << ra.upper() << std::endl;
        ra = ra + 3.0;
        std::cout << ra.lower() << ',' << ra.upper() << std::endl;
        return 0;
}
```

このプログラムを実行すると、[10,90] だった領域が、確かに両端が 3 ずつ拡大して [7,93] になったことがわかるでしょう。

同様に double x との減算（-）もできますし、乗算は中心位置から範囲を double x 倍にするとか、逆に除算は中心位置のまわりに double x 分の 1 に縮小するとかは簡単にできます。

さて、この実装では

```
        ra = ra + 3.0;
```

の部分を

```
        ra = 3.0 + ra;
```

とすると、エラーになります。これは + 演算子の右オペランドに Range クラスオブジェクトがくる演算が定義されていないからです。しかしメンバ関数として実装する限り、左オペランドが Range クラスオブジェクトです。右オペランドに Range クラスオブジェクトがくる場合にも対処するためには、メンバ関数ではなく、通常の関数として実装します。これについては次の整形入出力での << 演算子や >> 演算子の実装のところで述べましょう。

8.2 クラスオブジェクトの整形出力

演算子 << や >> も多重定義可能な演算子です。

もともと C++ は C の大部分をそのまま引き継いでますので、演算子 << と >> の本来の意味は左シフト演算子と右シフト演算子で、オペランドは整数型か列挙型です。しかし、今まで多くの例題で見てきたように、左オペランドには標準入力や標準出力が使えています。またこの時、右オペランドは整数型のみならず、浮動小数点数型や文字列型なども使えています。これは入出力ライブラリの中で演算子 >> や<< を多重定義しているからに他なりません。

8.2.1 フレンド指定子

演算子をオーバーロードする場合、メンバ関数でなければならない場合と、メンバ関数としなくてもよい場合があります。メンバ関数として定義する場合、そのオブジェクトは常に左オペランドでなければなりません。ところが、今の場合、左オペランドは標準出力ストリームのオブジェクトか標準入力ストリームのオブジェクトなので、クラスのメンバ関数として定義することはできません。幸いにも演算子 >> と<< はメンバ関数としなくてもよい演算子ですので、クラス定義の外でこの演算子をオーバーロードします。

ところが、ここで問題が生じます。入力や出力を扱うわけですので、クラスオブジェクトのメンバ変数の読み書きが必要ですが、クラスオブジェクトの外からは private や protected のアクセス指定がされたメンバ変数にはアクセスできません。しかし、これを public にしてしまうと、外からアクセス可能になり、private や protected にした意味がありません。

このような場合の解決策を与えるのがフレンド指定子（friend 指定子）です。フレンド指定子のつけられた関数をフレンド関数、クラスをフレンドクラスと呼びます。演算子は関数形式で扱うことができますので、フレンド関数に含まれます。

フレンド関数やフレンドクラスは、アクセスを受ける側のクラス定義の中で宣言します。

では、実際に我々の mylibrary::Range クラス定義の中で、まずは整形出力を表す演算子 << をフレンド関数として宣言しましょう。

```
class Range
{
friend std::ostream& operator<<(std::ostream& os, const Range& range);
        :
```

のように書きます。このように宣言すると、この関数（演算子）は Range クラスオブジェクトの private メンバや protected メンバにアクセスできます。

もちろん、関数の宣言も必要です。メンバ関数ではありませんので、クラス Range の定義の外で

```
std::ostream&
operator<<(std::ostream& os, const Range& range);
```

を宣言します。

8.2.2 整形出力の実装

これで準備ができたので、簡単な実装をしてみます。

```
std::ostream&
operator<<(std::ostream& os, const Range& range)
{
        os << '[' << range.m_lower << ',' << range.m_upper << ']';
        return os;
}
```

実際に使用してみます。

```
#include <iostream>
#include "range.h"

int main()
{
        mylibrary::Range range1(10.0,90.0), range2(20.0,80.0);
        std::cout << "range1=" << range1 << std::endl;
        std::cout << "range2=" << range2 << std::endl;
        return 0;
}
```

8.3 クラスオブジェクトの整形入力

整形入力を行う >> 演算子についても整形出力の場合とほぼ同じやりかたで実装できます。ただし、入力はオブジェクトの値を変更しますので、引数の型は const Range& ではなく Range& になります。出力と入力、両方書くとヘッダは

```
class Range
{
friend std::ostream& operator<<(std::ostream& os, const Range& range);
friend std::istream& operator>>(std::istream& is, Range& range);
        :
};
std::ostream&
operator<<(std::ostream& os, const Range& range);
std::istream&
operator>>(std::istream& is, Range& range);
```

プログラム 8.1: 整形入出力付き Range class（ヘッダ）

```
1  #ifndef RANGE_H_INCLUDED
2  #define RANGE_H_INCLUDED
3  #include <iostream>
4  namespace mylibrary
5  {
6
7  class Range
8  {
9  friend std::ostream& operator<<(std::ostream& os, const Range& range);
10 friend std::istream& operator>>(std::istream& is, Range& range);
11
12 public:
13         Range();
14         Range(double lower, double upper);
15         double lower() const { return m_lower; };
16         double upper() const { return m_upper; };
17         bool is_inside(double x) const;
18 private:
19         double m_lower;
20         double m_upper;
21 };
22
23 std::ostream&
24 operator<<(std::ostream& os, const Range& range);
25 std::istream&
26 operator>>(std::istream& is, Range& range);
27
28 } // namespace mylibrary
29 #endif // RANGE_H_INCLUDED
```

のようになります。

さて、実装では少し考察が必要です。出力の場合は、ここで示した実装のように出力オブジェクトのエラーなどお構い無しに出力しても出力オブジェクトがエラー状態になるだけで余り問題はありません。しかし入力の場合、そうは簡単にはいきません。例えば、最初の [が来るべきところで、それ以外の文字が入力された場合、文字入力要求に対し文字が入力されたので、入力装置は正常状態のままです。しかし Range オブジェクトの値の読み取りとしてはエラーです。もし、この時点で何もせずに入力処理関数から戻ってしまうと、入力にエラーがあったこと、すなわち Range オブジェクトの値が正しく読み取れなかったことがわかりません。これに対処するために、ここの実装では std::istream のメンバ関数（実際には基底クラスのメンバ関数を継承したもの）である setstate を使って入力ストリームオブジェクトの状態をエラー状態に設定しています。

プログラム 8.2: 整形入出力付き Range class（1/2）

```
1  // range.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "range.h"
5
6  namespace mylib
7  {
8
9  Range::Range() :
10         m_lower(0.0), m_upper(0.0)
11 {
12 }
13
14 Range::Range(double xmin, double xmax) :
15         m_xmin(xmin), m_xmax(xmax)
16 {
17 }
18
19 bool Range::is_inside(double x) const
20 {
21         if((x < m_lower) || (x > m_upper))
22                 return false;
23         return true;
24 }
25
26 std::ostream&
27 operator<<(std::ostream& os, const Range& range)
28 {
29         std::streamsize w = os.width();
30         std::streamsize hw;
31         if (w < 5)
32                 hw = 0;
33         else
34                 hw = (w - 3) / 2;
35         os << std::setw(0) << '['
36                 << std::setw(hw) << range.m_xmin << ','
37                 << std::setw(hw) << range.m_xmax << ']';
38         return os;
39 }
```

プログラム 8.3: 整形入出力付き Range class（2/2）

```
1  std::istream&
2  operator>>(std::istream& is, Range& range)
3  {
4          char c;
5          double xmin;
6          double xmax;
7
8          if (!(is >> c))
9                  return is;
10         if (c != '[')
11         {
12                 is.setstate(std::ios::failbit);
13                 return is;
14         }
15         if (!(is >> xmin))
16                 return is;
17         if (!(is >> c))
18                 return is;
19         if (c != ',')
20         {
21                 is.setstate(std::ios::failbit);
22                 return is;
23         }
24         if (!(is >> xmax))
25                 return is;
26         if (!(is >> c))
27                 return is;
28         if (c != ']')
29         {
30                 is.setstate(std::ios::failbit);
31                 return is;
32         }
33         range.m_xmin = xmin;
34         range.m_xmax = xmax;
35         return is;
36 }
37
38 }       // namespace mylibrary
```

第9章　派生と継承

あるクラスに、いくつかの機能を付け加えたクラスを作りたくなることはよくあります。このような場合には、元のクラスをそのまま使って新しい機能だけを追加して新しいクラスを作ることができます。こうして作られた新しいクラスを派生クラス（derived class）と言い、元のクラスを基底クラス（base class）と言います。この時、派生クラスは基底クラスの属性を引き継ぎますが、その関係を継承（inheritance）と呼びます。

ここでは、今まで作ってきた Range クラスを例にこれらの使い方を例示しましょう。

9.1　派生クラスを作る

ある一連の数値データがあって、ある範囲内に入ったデータ数（あるいは範囲に入らなかったデータ数）を知りたいとします。いままで作ってきた我々の Range クラスの、is_inside() を使って true の場合だけカウント数を 1 だけ増加させればよいわけです。この機能がプログラム中の非常に限られた箇所でしか使われないのであれば、特に新しいクラスを作らずに、そのままコードを記述してもよいでしょう。しかし、いろいろな箇所で使われるのであれば、その機能を Range クラスに押し込めるのも一つの解です。そうすることで、Range と、その範囲に入ったデータのカウントという機能が一体化され、何箇所もあるカウントを増やす場面で範囲の相手を間違えたり、カウントする変数を間違えたりというエラーを防ぐことができます。

ここでは我々の Range クラスから RangeCounter というクラスを派生さることにしましょう。これを行なうには、以下のように記述します。

```
class RangeCounter : public Range
{
        :
}
```

このように記述することにより、Range の public 属性を持つすべてのメンバが RangeCounter の public メンバとして扱うことができるようになります。

9.1.1　コンストラクタ

我々の Range クラスは二つの引数（範囲の最小値と最大値）を持つコンストラクタがあり、単一オブジェクトを生成する場合は、主にこちらを用いてきました。ここでも同様に RangeCounter クラスに二つの引数（範囲の最小値と最大値）を持つコンストラクタを定義しましょう。この時、基底クラスは二つの引数を持つコンストラクタで初期化することが求められます。これを指定するには、初期化リストで以下のように基底クラスのコンストラクタを明示します。

```
RangeCounter::RangeCounter(double rmin, double rmax) :
        Range(rmin, rmax)
{
```

```
        :
}
```

9.1.2 メンバの追加

ここで作る RangeCounter クラスは、範囲内に入ったデータの個数を数えるものです。まずは、これを実現しましょう。

個数を記憶させる必要がありますので、それをメンバ変数 int m_inside として導入します。また範囲内に入った数を数える関数が必要ですので、それをメンバ関数を count(double x) としましょう。この関数の定義は

```
int RangeCounter::count(double x)
{
        if (is_inside(x))
                m_inside = m_inside + 1;
        return m_inside;
}
```

で出来上がりです。この if 文の中で is_inside() をあたかも自分のメンバ関数のように使っています。Range クラスから継承されているので、このような使い方ができます。

もし RangeCounter クラスの方でも is_inside() を定義していると、そちらが優先されます。しかし、この関数の中で必要なのは Range クラスの is_inside なので、その場合にはクラスを明示して

```
int RangeCounter::count(double x)
{
        if (Range::is_inside(x))
                m_inside = m_inside + 1;
        return m_inside;
}
```

のようにします。つまりクラス名は名前空間の名前と同様の役割も果たします。今の場合はクラス名無しでも正しく機能しますが、将来、同じ名前の関数を RangeCounter でも用意する可能性が少しでもあるなら、この例のようにクラス名までつけて関数を指定しておくのがよいでしょう。

なお、

```
                m_inside = m_inside + 1;
```

の部分は、

```
                m_inside += 1;
```

とも書けますし、

```
                m_inside++;
```

や

```
                ++m_inside;
```

とも書けます。今の場合、どれも同じ結果を与えますので、どれも正解です。ただ、実行速度を考えた場合は一番最後の形が最も効率的である可能性が高くなります。可能性という言い方をしたの

は、どのような最適化が行なわれるかはコンパイラに依存するからです。単純に翻訳すれば、最初の二つは右辺の評価と代入が行なわれますが、後者二つは（対応する命令があれば）記憶域にある値を直接増加させるので実行速度的に有利になります。また後者二つの比較では後置の++は、式の値は増加前の値ですので、その値をどこかに保存しておかなければなりませんが、前置の場合の式の値は増加後の値ですので、増加前の値を保存する必要はありません。従って、一番最後の書き方が最も効率的である可能性が高くなるというわけです。

9.1.3 初期化を考える

さて、m_inside の初期化をどこで行なうかという問題を考えましょう。これには、大きく分けて二つの考え方があります。

一つは、この RangeCounter クラスはカウンタを表すものであるという考え方です。オブジェクトは生成されると直ちに使うことができて使い終えたら解放するという考え方です。この考え方では、初期化はコンストラクタで行うことになります。この考え方の利点は同じオブジェクトでのカウントがプログラムのあちらこちらで使われていても誤ってカウンタをリセットすることは絶対に起こらないということです。一方、リセットして新たにカウントしたい時には同じ範囲であっても別のオブジェクトを生成しなければなりません。これはプログラムによっては不便になります。

もう一つの考え方は、このクラスはあくまでも Range クラスの拡張であり、カウントするという機能が付け加わったものであるとする考え方です。つまり Range の一種であって、そのオブジェクトはカウントするしないにかかわらず Range が存在する限り存在するという考え方です。この場合は m_inside は単に RangeCounter オブジェクトの状態を表す変数の一つということで、コンストラクタで初期化されるほか、この変数を初期状態に戻す関数を使うことが可能であるように設計することになります。こうすると、プログラムのどこかで誤って初期状態に戻してしまう危険性が生じることになります。その一方で、同じ Range でカウントし直すことが可能になります。

実際のプログラムでは、後者を実現できれば前者へは簡単に修正できます。例えば m_inside を初期化する関数を void reset() とするならば、後者であれば public とし、前者にするなら private とすれば実現できます。

9.1.4 限定公開

派生クラスで機能を付け加えるときに、基底クラスのメンバを使う必要が生じることがあります。例えば、すぐに思いつく拡張として、指定された範囲のカウント数の他に下限値未満だった個数や上限値を越えた個数も数えたいというのがあるでしょう。これを実現するのも論理的には難しい話ではありません。新たにこれらを数える変数 m_under と m_over を付け加え、count() の中で下限値未満の場合、上限値より大きい場合の処理を付け加えるだけです。例えば

```
int RangeCounter::count(double x)
{
        if (is_inside(x))
        {
                ++m_inside;
                return m_inside;
        }
        else if (x <= m_lower)
                ++m_under;
        else if (x >= m_upper)
```

```
                ++m_over;
        return 0;
}
```

とすればよいでしょう。元の論理を変えないように範囲に入ったらカウントを 1 増やして、その値を戻し、そうでなければ 0 を戻すようになっています。また、ここで is_inside() の判定を先に行い、残りの判定で等号が含まれていることにも注意します。これにより is_inside() の判定で端点が含まれていても、あるいは端点が含まれていなくても正しく動作します。つまり将来 is_inside() の判定で端点処理が変更になっても対応できる論理になっています。

ところが、ここで問題が生じます。上限値、下限値は基底クラスの限定（private）メンバ変数です。派生クラスからはアクセスできません。これを公開（public）メンバ変数に変更すればアクセスできるようになりますが、それでは限定メンバ変数に指定した意味がありません。このような場合、派生クラスにのみアクセスを許すアクセス指定子、protected （限定公開）を用います。このアクセス指定子が付けられたメンバ変数やメンバ関数は、派生クラスからはアクセスできますが、派生していないクラスからはアクセスできません。そこで Range クラスの m_upper と m_lower のアクセス指定子を以下のように protected に変更します。

```
class Range
{
        :
protected:
        double m_lower;
        double m_upper;
};
```

9.1.5 定義例

ここまでで、新たなクラス RangeCounter を定義できる準備が整いました。定義例を示します。

プログラム 9.1: RangeCounter のヘッダ

```
1  #ifndef RANGECNT_H_INCLUDED
2  #define RANGECNT_H_INCLUDED
3  #include "range.h"
4  namespace mylibrary
5  {
6
7  class RangeCounter : public Range
8  {
9          unsigned int m_under;
10         unsigned int m_over;
11         unsigned int m_inside;
12 public:
13         RangeCounter();
14         RangeCounter(double lower, double upper);
15         void reset();
16         bool count(double x);
17         unsigned int count() const {return m_inside;}
18         unsigned int under() const {return m_under;}
19         unsigned int over() const {return m_over;}
20 };
21
22 }       // namespace mylibrary
23 #endif  // RANGECNT_H_INCLUDED
```

プログラム 9.2: RangeCounter の定義

```
1  #include "rangecnt.h"
2  namespace mylibrary
3  {
4
5  RangeCounter::RangeCounter()
6  {
7          reset();
8  }
9
10 RangeCounter::RangeCounter(double lower, double upper) :
11         Range(lower, upper)
12 {
13         reset();
14 }
15
16 void
17 RangeCounter::reset()
18 {
19         m_under = 0;
20         m_over = 0;
21         m_inside = 0;
22 }
23
24 bool
25 RangeCounter::count(double x)
26 {
27         if (is_inside(x))
28         {
29                 m_inside++;
30                 return true;
31         }
32         else if (x <= m_lower)
33         {
34                 m_under++;
35                 return false;
36         }
37         else if (x >= m_upper)
38         {
39                 m_over++;
40                 return false;
41         }
42         return false;
43 }
44 }       // namespace mylibrary
```

9.2 仮想関数

派生クラスにおいて、その基底クラスから継承したメンバ関数は、基底クラスのポインタや参照を使ってアクセスすると、基底クラスのメンバ関数が呼び出されます。しかし基底クラスのポインタや参照から、その派生クラスのメンバ関数が呼び出せたら便利なことがあります。これを実現するのが仮想関数です。

9.2.1 基底クラスで管理する

例えば今、いくつかのスロットを持つ汎用計測システムがあったとします。各スロットにはユーザが様々なモジュールを挿して、目的に応じて計測システムを組みあげることができるものとしましょう。モジュールには電流を測るモジュール、電圧を測るモジュール、温度を測るモジュールがあるとします。これをプログラムで制御したりデータを読み取るとき、Module というクラスを導入し、

- 電流を測るモジュールは Current クラス、
- 電圧を測るモジュールは Voltage クラス、
- 温度を測るモジュールは Temperature クラス

として、この各々は Module クラスから派生させるとしましょう。すべてのクラスには name() というメンバ関数があり、以下のように C 型文字列のポインタを返すものとします。

```
class Module
{
public:
        const char* name() { return "Module"; }
};

class Current : public Module
{
public:
        const char* name() { return "Current"; }
};

class Voltage : public Module
{
public:
        const char* name() { return "Voltage"; }
};

class Temperature : public Module
{
public:
        const char* name() { return "Temperature"; }
};
```

今、電流モジュールを 3 つ、電圧モジュールを 2 つ、温度モジュール 1 つの 6 つのモジュール使ってシステムを組みあげたとします。この 6 つのモジュールをプログラムで表現するために配列を使いたいところですが、型が異なるために、簡単な配列にはできません。しかしいずれも Module の派生クラスなので、これを表現するのに以下のように Module オブジェクトへのポインタ 6 個の配列とすることはできます。

プログラム 9.3: 基底クラスのポインタを使う

```
1  #include <iostream>
2  class Module
3  {
4  public:
5          const char* name() { return "Module"; }
6  };
7
8  class Current : public Module
9  {
10 public:
11         const char* name() { return "Current"; }
12 };
13
14 class Voltage : public Module
15 {
16 public:
17         const char* name() { return "Voltage"; }
18 };
19
20 class Temperature : public Module
21 {
22 public:
23         const char* name() { return "Temperature"; }
24 };
25
26 int main()
27 {
28         Current ct1, ct2, ct3;
29         Voltage vt1, vt2;
30         Temperature tm;
31         Module* slot[6]
32                 = {&vt1, &tm, &ct1, &ct3, &vt2, &ct2};
33         for(int i = 0; i < 6; ++i)
34                 std::cout << "Slot[" << i << "]␣"
35                         << slot[i]->name() << "\n";
36         return 0;
37 }
```

ところが、各スロットに入っているモジュールの名前を表示しようとして

```
        for(int i = 0; i < 6; ++i)
                std::cout << "Slot[" << i << "]␣"
                        << slot[i]->name() << "\n";
```

としても、呼び出されるのは基底クラスの `name()` 関数です。これらのポインタが派生クラスの基底部であるなら、派生クラスの `name()` 関数を呼び出してほしいところです。これを実現するのが仮想関数です。

9.2.2 仮想関数を宣言する

仮想関数であることを宣言するには基底クラスの関数宣言で `virtual` 修飾子を付与します。いまの場合は

```
class Module
{
```

```
public:
        virtual const char* name() { return "Module"; }
};
```

とします。この virtual は継承されます。つまり派生クラスに同じ名前で同じ引数の関数があれば virtual 修飾子が無くても virtual として扱われます。いまの場合は、上の宣言だけで Current クラス、Voltage クラス、Temperature クラスの name メンバ関数は virtual になります。仮想関数は、継承先を調べ、継承先に同じ名前で同じ引数を持つ関数が定義されていればその関数を使います。多段に継承されている場合は、次々にたどっていって最後に定義されている関数を使います。こうして基底クラスのポインタや参照から派生先のメンバ関数を呼び出すことができるようになります。

```
Slot[0] Voltage
Slot[1] Temperature
Slot[2] Current
Slot[3] Current
Slot[4] Voltage
Slot[5] Current
```

となります。

9.2.3 仮想デストラクタ

デストラクタも仮想にすることができます。これを宣言するには、基底クラスの宣言でデストラクタに virtual 修飾子をつけて宣言します。

仮想デストラクタは非常に重要です。基底クラスのポインタだけで派生先まで含めて delete することができるからです。仮想デストラクタであれば、派生先から順に delete していきます。

同一の基底クラスから派生した様々なクラスの多数のオブジェクトを管理する際に、9.2.1 の例で見たように基底クラスのポインタで管理できると非常に楽になります。解放する場合も同様です。

もし仮想デストラクタでない場合、基底クラスのポインタを delete すると基底クラスの部分だけが解放され、派生クラスのオブジェクトは非常におかしな状態になります。通常、基底クラスとなる可能性のあるデストラクタは必ず仮想デストラクタにします。

9.2.4 純粋仮想関数

仮想関数は派生クラスで定義しない場合、基底クラスの定義が使われますが、派生クラスで必ず定義させたいという場合もあります。

例えば 9.2.1 で、Module の name() ではとりあえず "Module" という文字列へのポインタを返してますが、これが使われるべきではなく、派生クラスでの name() が使われるべきでしょう。つまり基底クラスの name() 関数は、各派生クラスでこの関数があるべきだということを示しているだけで、定義は各派生クラスで行われるべきものです。これを示すには仮想関数を = 0 として与えます。今の例では

```
class Module
{
public:
```

```
        virtual const char* name() = 0;
};
```

と書き、このような関数を純粋仮想関数と呼びます。純粋仮想関数は派生クラスで定義を与える必要があります。逆に言えば純粋仮想関数を一つでも持つクラスはそのクラスオブジェクトを作ることができません。このようにクラスオブジェクトを作ることができないクラスを抽象クラスと呼ぶことがあります。

第10章　オブジェクトの動的生成と解放

ここまでのやり方では、オブジェクトは定義により生成され、その実行ブロックから出ると自動的に解放されます。この機構は便利で安全ですが、実行の状況に応じて配列の大きさが変わる場合や、必要とされるオブジェクトが異なる場合などでは必要に応じて動的にオブジェクトを生成し、不要になったと時に解放できると便利です。

10.1　動的生成と解放に関する基本事項

C++ では、new 演算子と delete 演算子を使って、動的なオブジェクトの生成と解放を行うことができます。注意すべきは、この二つの演算子は必ず対で使うという点です。new 演算子で生成したオブジェクトは必ず delete 演算子で解放しなければなりません。ここでは、いくつかの例を用いて、これらの使い方を述べます。

10.1.1　単一オブジェクトの動的生成と解放

単一のオブジェクトを動的に生成するには new 演算子を使います。この演算子は右オペランドに型名をとり、その指定された型のオブジェクトを記憶域に生成し、そのポインタを返します。例えば

```
int * pint = new int;
```

とすると、整数を格納する記憶域を生成し、そのポインタが pint に格納されます。生成時に初期値を設定することもできます。

```
int * pint = new int(3);
```

とすると、*pint は 3 になります。

解放するには delete 演算子を使います。上の例で生成した記憶域を解放するには

```
delete pint;
```

とします。

もちろん、クラスオブジェクトを生成することもできます。いままで作ってきた、我々の Range オブジェクトは

```
mylibrary::Range * pr = new mylibrary::Range;
```

とすることで生成できます。この時は省略時コンストラクタが呼ばれますが、

```
mylibrary::Range * pr = new mylibrary::Range(10.0, 90.0);
```

とすると、コンストラクタ Range(double, double) が呼ばれます。解放するのも int の時と同様

```
        delete pr;
```

です。

10.1.2 配列の動的生成と解放

配列に対しても同様に動的に生成し不要になった時点で解放することができます。

```
        int * piarray = new int[100];
```

とすると、100 個の整数を要素とする配列が生成され、そのポインタが piarray に格納されます。これを解放するには

```
        delete[] piarray;
```

とします。単なる delete ではなく、delete[] であることに注意してください。

C++ のオブジェクトは一定の記憶域を占めるということは今まで何度も述べてきました。new 演算子で生成したオブジェクトは delete 演算子で解放するまで記憶域に残り続けます。下手な使い方や間違った使い方をすると、不必要なオブジェクトがいつまでも記憶域を占有するという事態になります。これが、繰り返しの回数の多いループの中で起こるとたちどころに記憶域を食いつぶすという事態にもなります。ところが、new と delete は対で使わなければならないにも係わらず、構文的に、この二文が並ぶことはまずありません。生成して使わずにすぐに解放するのであれば最初から生成する必要が無いということです。つまり、new で生成するからには、それに対応する delete は離れた位置にあるのが普通です。これが delete を忘れたり、間違った相手を delete する誘因となります。

これを防ぐ一つの方法は、直接 new や delete を使うことは止めて、クラスの中に押し込めてしまうことです。この場合、問題の複雑さをクラスに押し込めることになるので、クラスの設計はしっかり行う必要があります。標準ライブラリには動的生成のためのクラスが用意されています。使えるところでは、できるだけ標準ライブラリを使うのがよいでしょう。

10.2 動的生成を伴うクラス

演算子 new による動的生成では演算子 delete による解放が必要です。この対を誤って使用したり、new したものの delete を忘れるといった誤りを防ぐために、この対をクラスに閉じ込めるというのは一つの方法です。しかしながら、この方法には、いくつか重要な注意点があります。ここでは、C 型文字列（文字配列）を格納するクラスを作りながら、その注意点を述べていきます。

10.2.1 文字配列を格納するクラス

C 型文字列（文字配列）を格納するクラスは標準ライブラリの vector で実現できます。また文字列として扱うのであれば、標準ライブラリの string を使うこともできます。実際のプログラムでは、効率や安全性からも、これらを使うべきですが、ここでは練習のため、この文字配列を格納するクラスを CharArray として自作してみましょう。

メンバ変数、コンストラクタとデストラクタ

配列を new を使って動的に割り当てますが解放するときにはそのポインタが必要ですので、そのポインタを保持しておく必要があります。その変数名を m_parray としましょう。省略時コンストラクタで、これを 0 に初期化しておきます。また、デストラクタでは、delete 演算子を使って解放しますが、0 ポインタに対しても delete 演算子は正しく働くので、この時点でデストラクタも定義しておきましょう。また、標準出力への整形出力があるとテストなどに便利ですので、これも宣言しておきましょう。

```
#include <iostream>
class CharArray
{
friend std::ostream& operator<<(std::ostream& os, const CharArray& ca);
public:
        CharArray() : m_parray(0) {}
        CharArray(const char* str);
        ~CharArray() { delete [] m_parray; }
private:
        char* m_parray;
};
std::ostream& operator<<(std::ostream& os, const CharArray& ca);
```

整形出力

ここでの整形出力は C 型の文字列を出力するだけです。ヘッダでこの演算子は friend 宣言してあるので、CharArray の private メンバ変数である m_parray にアクセスできます。

```
std::ostream& operator<<(std::ostream& os, const CharArray& ca)
{
        if (ca.m_parray)
                os << ca.m_parray;
        return os;
}
```

引数をとるコンストラクタ

省略時コンストラクタは、すでにヘッダで与えたので、ここでは引数をとるコンストラクタを定義します。C 型文字列用のライブラリを使うのが便利ですので、<cstring> をインクルードしておきます。

```
#include <cstring>
        :
CharArray::CharArray(const char* str) :
        m_parray(0)
{
        if(str)
        {
                m_parray = new char [strlen(str) + 1];
                strncpy(m_parray, str, strlen(str) + 1);
        }
}
```

C 型の文字列は終端に文字 0 を置きますので、配列の大きさは文字数に 1 を加える必要があります。new を使って文字配列を確保した後、最後の 0 まで含めてコピーしています。

テストしてみる

一通り準備ができたので、テストしてみます。ヘッダファイルを chararray.h として、以下のようなプログラムを書くと、標準出力に hello, world と出力されます。

```
#include <iostream>
#include "chararray.h"
int main()
{
        CharArray ca = "hello, ";
        CharArray cb = "world";
        std::cout << ca << cb << std::endl;
        return 0;
}
```

これで一見完成したように思えますが、実は重大な問題が隠されています。以下では、その点について見てましょう。

10.2.2 コピーコンストラクタ

さて、10.2.1 で導入した CharArray クラスを、もう少しテストしてみます。

繰り返し文の最初に同じクラスのオブジェクトを使って初期化することはよくあります。繰り返しの中のどこかで別の値を設定する可能性がある場合などです。そこで、以下のようなプログラムを作ってみます。

```
#include <iostream>
#include "chararray.h"
int main()
{
        CharArray ca = "start ";
        for (int i = 0; i < 5; i++)
        {
                CharArray cb = ca;
                std::cout << cb << i << std::endl;
        }
        return 0;
}
```

このプログラムは、繰り返しの最初に CharArray クラスのオブジェクト cb を、同じクラスのオブジェクト ca を初期値として生成しています。ところが、このプログラムを実行すると何が起こるか予想つきません。筆者の環境では

```
E:\cppintro>a
hello,

#J
#J
#J
```

のような出力になってしまいました。

実は、これまでの定義だけでは new と delete の対が合わなくなります。そうならないようにクラスに閉じ込めたはずなのにです。

自動生成されるコピーコンストラクタ

ここでの問題は

```
        CharArray cb = ca;
```

にあります。この時、新たに CharArray クラスのオブジェクトを生成するので処理系はコンストラクタを呼び出します。この時、同じクラスのオブジェクトが初期値として与えられているので、処理系は同じクラスのオブジェクトを引数に持つコンストラクタを探します。この、同じクラスのオブジェクトを引数に持つコンストラクタをコピーコンストラクタと呼びます。処理系は、もしそれが与えられていなければ、メンバ変数の値を全部コピーするコンストラクタを自動的に生成します。これにより、クラスを定義する時に、いちいちコピーコンストラクタを定義する手間が省けます。また C の構造体をクラスの一形態として扱うことができます。

ところが、我々の CharArray クラスではポインタの値がコピーされることになるので、同じアドレスを二つのオブジェクト（今の場合 ca と cb）が持つことになります。従って、片方のオブジェ

クト（例えば cb）が解放された時点で、そのポインタは無効のポインタになり、もう片方（ca）のポインタも（同じ値ですので）無効になります。そうなった時点で残っているオブジェクト（ca）の値を参照すると、無効のアドレスを参照することになります。

コピーコンストラクタを定義する

ここでの問題を防ぐには、コピーコンストラクタを定義する必要があります。
まずヘッダにコピーコンストラクタの宣言を追加します。

```
        CharArray(const CharArray& ca);
```

次に定義ですが、これは const char* を引数に持つコンストラクタと同じように書けます。

```
CharArray::CharArray(const CharArray& ca) :
        m_parray(0)
{
        if(ca.m_parray)
        {
                m_parray = new char [strlen(ca.m_parray) + 1];
                strncpy(m_parray, ca.m_parray, strlen(ca.m_parray) + 1);
        }
}
```

10.2.3 代入演算子

自動生成されるコピーコンストラクタによって、new と delete の対関係が壊される問題には対処できましたが、まだ問題が残っています。テストプログラムをほんの少し変えてみます。

```
#include <iostream>
#include "chararray.h"
int main()
{
        CharArray ca = "hello, ";
        for (int i = 0; i < 5; i++)
        {
                CharArray cb;
                cb = ca;
                std::cout << cb << std::endl;
        }
        return 0;
}
```

前との違いは、繰り返しの中で、オブジェクトの生成には省略時コンストラクタを使い、代入で値を設定していることです。筆者の環境では、ここで再びわけのわからない出力になってしまいました。

自動生成される代入演算子

この原因は、

```
        cb = ca;
```

にあります。これは値の代入ですが、代入演算子が定義されていない場合、処理系は自動的に、すべてのメンバ変数の値を代入する代入演算子を定義します。つまり、いまの場合

```
cb.m_parray = ca.m_parray;
```

という演算が行なわれます。ポインタの値がコピーされるので、コピーコンストラクタ (10.2.2) で述べた問題が再び起こります。これを防ぐには、代入演算子を定義します。

代入演算子を定義する

代入演算子は通常、以下のように宣言します。

```
CharArray& operator=(const CharArray& ca);
```

代入演算子では、それまで保持していた資源を解放してから、コピーコンストラクタと同様の機能を果たします。ただし、自分自身への代入ということも有りえますので、このチェックを先に行い、もしそうなら、そのまま何もせず自分自身の値を返します。

```
CharArray& CharArray::operator=(const CharArray& ca)
{
        if (ca.m_parray == m_parray)
                return *this;
        delete [] m_parray;
        m_parray = 0;
        if(ca.m_parray)
        {
                m_parray = new char [strlen(ca.m_parray) + 1];
                strncpy(m_parray, ca.m_parray, strlen(ca.m_parray) + 1);
        }
        return *this;
}
```

これでようやく安全に使えるようになりました。

10.2.4 コピーコンストラクタや代入演算子使用の禁止

ここまで見てきたように、動的生成を伴うクラスでは一般に

- デストラクタ
- コピーコンストラクタ
- 代入演算子

を与える必要があります。これは記憶域に限らず、動的に資源を確保するクラスについて一般的に言えることです。しかし場合によっては、このような操作を禁止してしまいたい場合があります。

これを行なうには宣言だけ行なって定義を与えないようにします。宣言が与えられることにより、処理系はそれらを自動的に生成することはなくなります。また宣言だけ与えられていて定義が与えられていなくても、プログラム中でそれが使われていなければエラーにはなりません。逆に使われていると、自動生成されることなく未定義関数（演算子）を使用した旨のエラーになります。こうしてコピーコンストラクタや代入演算子の使用を禁止することができます。

第11章　例外処理

プログラム開発を分散して行う時に、関数やメンバ関数の中で実行中に実行をそれ以上遂行できない条件に遭遇することがあります。この対処方法として、状況を示す変数を用意し、その値を実行不能に設定して終了し、この変数を呼び出し側にチェックさせるという方法があります。従来の C のライブラリ関数の大多数はこの方法を採っています。C++ では、この方法に加え、例外を投げる（throw）という手段が提供されています。呼び出された側が例外を投げると、制御は呼び出し側のプログラムブロックから一気に抜けて、その例外を捕捉（catch）するブロックへと制御が移ります。この方法により個々の関数呼び出し毎に結果をチェックするより見通しのよいプログラムが書ける場合があります。一方で、ブロックからブロックへの移動であるために、資源管理をしっかり行わないと、資源が解放されないままになる可能性があります。これらの点について、順に見ていきましょう。

11.1　例外オブジェクトとその捕捉

C++ での例外 (exception) は、例外を表現する何らかのオブジェクトです。C++ のオブジェクトであれば何でも構いません。この例外を捕捉 (catch) するには、そのオブジェクトの型を指定します。例を示しましょう。プログラム 11.1 は `double` 型変数の逆数を求める関数 `my_inverse()` の中で、変数 x が 0.0 の時に整数オブジェクトを投げています。これを使う `main()` の側では、例外が発生していない状況で実行すべきことを `try` ブロックで囲み、捕捉する例外の型を指定した `catch` ブロックを用意します。この `catch` ブロックを例外ハンドラ (handler) と呼びます。この `catch` ブロックの中で投げられたオブジェクトの値を必要とする場合は、その値を受け取る変数を用意します。プログラム 11.1 を値も受け取るように書き換えたものを プログラム 11.2 に示します。

11.1.1　複数の型の例外を処理する

例外オブジェクトは C++ のオブジェクトであれば何でも使えます。従って、例外を捕捉する側でも複数の型を捕捉する必要が出てくることがあります。プログラム 11.3 は、関数の中で x が 0.0 の時は整数型オブジェクトを、x が負の時には `double` 型のオブジェクトを投げます。これを区別して処理するには、各々の型に対し `catch` ブロックを用意します。プログラム 11.3 では `main()` の中で二つの `catch` ブロックを用意しています。

例外の型の判定は、書かれている順に行われます。つまり プログラム 11.3 ではまず `int` 型であるかどうかが判定され、そうでなければ次に `double` 型であるかどうかが判定されます。もし、どれにも捕捉されなければ、更にその外のブロックへ例外が伝播されます。

プログラム 11.1: 単純な例外処理の例

```
1  #include <iostream>
2
3  double my_inverse(double x)
4  {
5          if (x == 0.0)
6                  throw static_cast<int>(-1);
7          return 1.0 / x;
8  }
9
10 int main()
11 {
12         try
13         {
14                 double x;
15                 while (std::cin >> x)
16                         std::cout << my_inverse(x) << std::endl;
17         }
18         catch(int)
19         {
20                 std::cerr << "Exception: int" << std::endl;
21         }
22         return 0;
23 }
```

プログラム 11.2: 例外の値も捕捉する

```
1  #include <iostream>
2
3  double my_inverse(double x)
4  {
5          if (x == 0.0)
6                  throw static_cast<int>(-1);
7          return 1.0 / x;
8  }
9
10 int main()
11 {
12         try
13         {
14                 double x;
15                 while (std::cin >> x)
16                         std::cout << my_inverse(x) << std::endl;
17         }
18         catch(int n)
19         {
20                 std::cerr << "Exception: int " << n << std::endl;
21         }
22         return 0;
23 }
```

プログラム 11.3: 複数の型の例外を捕捉する

```
1  #include <iostream>
2  #include <cmath>
3
4  double my_sqrtinverse(double x)
5  {
6          if (x == 0.0)
7                  throw static_cast<int>(-1);
8          else if (x < 0.0)
9                  throw x;
10         return 1.0 / sqrt(x);
11 }
12
13 int main()
14 {
15         try
16         {
17                 double x;
18                 while (std::cin >> x)
19                         std::cout << my_sqrtinverse(x) << std::endl;
20         }
21         catch(int n)
22         {
23                 std::cerr << "Exception: int " << n << std::endl;
24         }
25         catch(double d)
26         {
27                 std::cerr << "Exception: double " << d << std::endl;
28         }
29         return 0;
30 }
```

11.1.2 例外の再送出

例外ハンドラの中から例外を投げることは可能です。一般に例外を投げるのは実行が不可能になったことを外のブロックへ通知し、後の処理を任せるためなので、例外を受け取った側では、自分の処理すべき部分を処理した後、同じ例外をそのままさらに外へ送出する必要がある場合があります。このような場合、例外ハンドラの中で、オブジェクトを指定せずに throw すると、捕捉した例外オブジェクトをそのまま外へ送出します。この時投げられる例外オブジェクトは、捕捉したのが、その基底クラスのオブジェクトとして捕捉した場合であっても、元の例外であることに注意します。つまり throw が投げる例外は、その基底クラスで捕捉した場合であっても、その例外ハンドラ内でのアクセスが基底クラスに限定されるだけで、再送出される情報が失われることはありません。

11.1.3 捕捉されなかった例外の処理

例外が捕捉されない場合、内側から順に呼び出した側の例外ハンドラを調べていきます。途中、どこにもその例外ハンドラが無い場合は、当然ながら関数 main にまで戻ってきます。ここにも対応する例外ハンドラが無い場合は、std::terminate() が呼び出されます。この関数の振る舞いはユーザにより変更可能です。既定値では abort() が呼び出されます。

11.1.4 すべての例外を捕捉する

例外オブジェクトの型によらず、すべての例外を捕捉したい場合があります。この場合は catch(...) と書きます。すでに述べたように例外ハンドラは、書かれている順に判定されますので、この catch ブロックは一番最後に置きます。この catch(...) ブロック以後に置かれた catch ブロックには絶対到達しないことになるからです。

11.1.5 非同期事象と例外

例外処理機構は、あくまでもプログラムの中で例外的な状況が起こった時に、そのプログラム内の別のブロックに制御を移すための機構です。プログラムの外で非同期に起こる事象に対する手段ではないことに注意します。例えばハードウェアによる割り込み信号を受けて制御を移すなどという機構とは全く別の機構です。従って例えば 0 による除算に対しシステムによっては単純に例外を使うことができないことがあります。

表 11.1: 標準例外

関連	クラス名	ヘッダ	要因
メモリ、型	bad_alloc	<new>	new
	bad_cast	<typeinfo>	dynamic_cast
	bad_typeid	<typeinfo>	typeid
	bad_exception	<exception>	exception-specification
入出力	ios_base::failure	<ios>	入出力ライブラリ
引数、添え字	out_of_range	<stdexcept>	範囲外
	invalid_argument	<stdexcept>	不正値
実行時エラー	overflow_error	<stdexcept>	オーバーフロー

11.2 標準例外

例外オブジェクトは C++ のオブジェクトであれば何でも使えるのですが、それでは余りにも一般的すぎて、捕捉する側では処理に困る場合があります。例外オブジェクトは系統的・階層的に分類できるようにしておかないと捕捉する側は大変です。

C++ の言語処理系や標準ライブラリが投げる例外を、標準例外と呼びます。標準例外はすべて <exception> で宣言されている exception クラスから派生したクラスのオブジェクトです。

基底クラスである exception クラスの公開メンバー関数は例外を投げません。従って、この公開メンバー関数を呼び出したことにより更に例外が発生することはありません。公開メンバー関数の中に仮想関数 what() という関数があります。この関数の戻り値は const char* 型で、この例外の情報を C 型文字列として与えることを想定しています。通常、派生クラスの方で実装されていますので、次のような使われ方がされるのが一般的です。

```
try
{
        // ...
}
catch (std::exception& e)
{
        std::cout << "std::exception: " << e.what() << std::endl;
        // ...
}
catch (...)
{
        std::cout << "Unknown exception" << std::endl;
        // ...
}
```

11.2.1 処理系や標準ライブラリが投げる例外

処理系や標準ライブラリによって投げられる例外は表 11.1 に示す 8 種類です。これらはいずれも exception クラスから派生しています。

表 11.2: 標準例外の階層構造

基底クラス		クラス名
exception	logic_error	length_error
		domain_error
		out_of_range
		invalid_argument
	runtime_error	range_error
		overflow_error
		underflow_error

11.2.2 標準例外の階層構造

標準ライブラリが投げる例外のみならず、<stdexcept> には表 11.2 に示す例外クラスが定義されています。これらは、実行する前に検出できる可能性のある論理的な例外クラス logic_error と、実行してみないと検出できない例外を表すクラス runtime_error に大別されています。独自の例外クラスを作る際には、この考え方を念頭に置いて、これらのクラスのどれかから派生させるのがよいでしょう。

11.3 オブジェクトの動的生成と例外

11.3.1 オブジェクトの動的生成と削除

オブジェクトを動的に生成するには new 演算子を使います。この演算子は生成するオブジェクトのコンストラクタを引数として持ち、生成したオブジェクトのポインタを返します。例えば、これまで述べてきた Range クラスのオブジェクトを動的に生成するには、

```
Range* pRange = new Range;
```

とします。上の場合は既定のコンストラクタが呼ばれますが、複数のコンストラクタを持つクラスの場合は、コンストラクタを明示することも可能です。例えば、

```
Range* pRange = new Range(-1.0,1.0);
```

とすると、double の引数を二つ持つコンストラクタが呼ばれます。new 演算子を用いて動的に生成されたオブジェクトは明示的に削除するまで記憶域に存在します。削除するには delete 演算子を用います。上の例では

```
Range* pRange = new Range(-1.0,1.0);
        :
delete pRange;
```

とします。これを忘れて先にポインタ変数を消したり、同じポインタ変数に別の値を代入したりすると、消せない記憶域ができてしまいます。例えば

```
if (...)
{
        Range* pRange = new Range(-1.0,1.0);
                :
}
```

のような構文の場合、if ブロックに入ると Range オブジェクトを new 演算子を使って生成することになりますが、pRange は if ブロックから抜けた途端に消去されてしまうので、if ブロックの中で delete しておかないと、消せない記憶域ができます。

消せない記憶域ができるもう一つの例を示しましょう。

```
Range* pRange;
for (int i = 0; i < 100; i++)
{
        pRange = new Range(-1.0,1.0);
                :
}
delete pRange;
```

この例の場合、for ブロックの中で delete を使って一回のループ毎に delete しておかないと、pRange の値がループ毎に置き換わっているので、for ループの外で消される記憶域は、最後の new で生成された記憶域のみ、つまり 100 回目の new で確保した記憶域のみです。その前 99 回の new で確保された記憶域が消されません。

11.3.2 配列の動的確保と削除

配列オブジェクトも動的に確保することができます。例えば整数 1000 個の配列を確保するには

```
int* parray = new int [1000];
```

とします。個数の指定には汎整数型の変数を用いることもできます。

これを削除する時は

```
delete [] parray;
```

とします。配列に対する new や delete には [] を伴うことに注意します。

11.3.3 オブジェクトの動的生成に伴う例外

オブジェクトの動的生成が失敗した場合、既定では std::bad_alloc の例外が投げられます。この例外クラスは、new で定義されています。<exception> ではないことに注意します。

通常、動的なオブジェクトの生成が失敗する理由は割り当て可能な記憶域が足りない場合です。例をプログラム 11.4 にしめします。この例は使用する環境に依存するので、block_size や num_blocks の値は適宜変更してください。プログラム中にある代入文や出力文は、最適化されて for ループが実行されなくなることを防ぐために入れてあるだけです。

筆者の環境（Windows 2000 SP4、512MB RAM、g++ (GCC) 3.4.5 (mingw-vista special r3)）では

```
C:\fujii\My Documents\cppintro>excep06
buf[0] allocated.
buf[1] allocated.
buf[2] allocated.
buf[3] allocated.
Exception caught: St9bad_alloc

C:\fujii\My Documents\cppintro>
```

となりました。

この時、注意しないといけないのは、try ブロックの中で new によって動的に確保したオブジェクトのポインタを記憶しておく変数は try ブロックの外にないと削除できなくなるという点です。例えばプログラム 11.5 のようにしてしまうと、例外が発生した途端に、それまでに動的に確保した記憶域を削除する手段が無くなります。この例では、動的割り当てに失敗したら単純に終了しますので、それまでに確保した記憶域は明示的に削除しなくても OS が解放することが期待できますが、もし続行する場合は、プログラム 11.4 のように、ポインタ変数 buf[] は try ブロックの外に置いて、必要に応じて削除できるようにしておかなくてはなりません。

プログラム 11.4: 動的生成の失敗を捕捉する

```
1  #include <new>
2  #include <iostream>
3
4  static const int block_size = 100000000;
5  static const int num_blocks = 100;
6
7  int main()
8  {
9          int *buf[num_blocks] = { 0 };
10         try
11         {
12                 for (int i = 0; i < num_blocks; i++)
13                 {
14                         buf[i] = new int [block_size];
15                         buf[i][0] = block_size;
16                         buf[i][block_size - 1] = i;
17                         std::cout
18                                 << "buf[" << i << "] allocated."
19                                 << std::endl;
20                 }
21                 std::cout
22                         << buf[num_blocks - 1][block_size - 1]
23                         << std::endl;
24         }
25         catch(std::bad_alloc& e)
26         {
27                 std::cerr
28                         << "Exception caught: "
29                         << e.what()
30                         << std::endl;
31         }
32         for (int i = 0; i < num_blocks; i++)
33                 delete [] buf[i];
34         return 0;
35 }
```

プログラム 11.5: 例外発生で記憶域が解放不能になる例

```
1          try
2          {
3                  int *buf[num_blocks] = { 0 };
4                  for (int i = 0; i < num_blocks; i++)
5                  {
6                          buf[i] = new int [block_size];
7                                  :
8                  }
9                          :
10                 for (int i = 0; i < num_blocks; i++)
11                         delete [] buf[i];
12         }
13         catch(std::bad_alloc& e)
14         {
15                 std::cerr << "Exception caught: " << e.what() << std::endl;
16         }
```

第12章　テンプレート

いままで見てきたように、C++ の関数やクラスのメンバ関数などでは同じ名前であっても引数の個数や型が異なれば、異なる関数として扱われます。一方、型に依らずに共通に存在する処理構造というものも存在します。C++ では、このような場合に効率的にコードを書く方法としてテンプレート（template）と呼ばれる機能を提供しています。これは、関数やクラス定義に対して、雛型（テンプレート）を書いておくと、コンパイラは、それを利用して必要な関数やクラスの具体的な形を自動生成するものです。

12.1　関数テンプレート

まずは、簡単な例として二つの引数のうち大きい方の値を返す関数を作ってみましょう。名前を `Larger` とします。この関数の定義は、例えば `double` を引数にするのであれば次のようになるでしょう。

```
double Larger(double x1, double x2)
{
        return (x2 > x1) ? x2 : x1;
}
```

では、`double` ではなく、`float` の場合はどうでしょう？更に `int` の場合はどうでしょう？型のところが違うだけで後は全く同ですね。このような場合、C++ では

```
template<class T>
T Larger(T x1, T x2)
{
        return (x2 > x1) ? x2 : x1;
}
```

と書いておいておいて、実際に使う時に例えば

```
        std::cout << Larger(3.14159, 2.71828) << std::endl;
```

などとすると、コンパイラは

```
double Larger(double x1, double x2)
{
        return (x2 > x1) ? x2 : x1;
}
```

という関数を生成してからコンパイルします。また、もし

```
        std::cout << Larger('G', 'U') << std::endl;
```

などとすると、

```
char Larger(char x1, char x2)
{
        return (x2 > x1) ? x2 : x1;
}
```

という関数を生成してからコンパイルします。つまり必要になった時に与えれらた型からテンプレートを使って実際の関数を作り出すわけです。

明示的な具現化

では、今までのテンプレートを使うとして、

```
        std::cout << Larger(3, 3.14) << std::endl;
```

の場合はどうなるでしょう？この場合は、int 型と double 型を引数にとる Large は無いのでエラーになります。ここで与えたテンプレートは二つの引数の型が一致している場合だけです。もし、どちらかを使いたいのであれば、その型を明示することで解決できます。

```
        std::cout << Larger<double>(3, 3.14) << std::endl;
```

と書くことで、

```
double Larger(double x1, double x2)
```

が使われます。このように、関数名の後に <> でくくって型名を与えることで、テンプレートから生成する関数を指定することができます。これを明示的な具現化と言います。

明示的な特殊化

さて、今までのテンプレートで

```
        std::cout << Larger("C++", "Java") << std::endl;
```

とするとどうなるでしょう？少なくともコンパイルでエラーは起こりません。なぜなら、二つの引数の型が一致しているからです。しかし結果は全く予測がつきません。文字列リテラルの型は const char* です。つまり、テンプレートから生成される関数は

```
const char* Larger(const char* x1, const char* x2)
{
        return (x2 > x1) ? x2 : x1;
}
```

です。これではポインタの大小比較をしていることになります。格納された番地のどちらが大きいかを調べているわけです。

もし Larger("C++", "Java") の意図が、辞書的な意味での文字列の比較であるならば（普通の意図はそうですよね)、これでは困ります。これを解決するには、const char* 型にだけ特別版を定義します。

```
template<class T>
T Larger(T x1, T x2)
{
        return (x2 > x1) ? x2 : x1;
```

```
}
template<> const char* Larger<const char*>
        (const char* x1, const char* x2)
{
        const char* t1 = x1;
        const char* t2 = x2;
        while (*x1 && *x2 && (*x1++ == *x2++));
        return (*x2 > *x1) ? t2 : t1;
}
```

この const char* に対する特別版は、C スタイルの文字列に対する標準関数の strcmp()（ヘッダ <cstring> で提供される）を使うことで単純化できますが、ここでは参考のために 1 文字ずつ比較し、どちらかに 0 が出るか、一致しなくなるまでポインタを進めていき、止まったところの、文字値の比較で返すポインタを選択するという手順を示してあります。

このように、特定の型に対し特別の版を用意することを明示的な特殊化と言います。

12.2 クラステンプレート

関数の場合と全く同様にクラスに対してもテンプレートを作ることができます。これをクラステンプレートと呼びます。ここでは、これまで作ってきた Range クラスをテンプレートにしてみましょう。また、Range クラスを利用する方のソースはそのままで済む工夫もしましょう。分担して開発する場合にはクラスを最初は通常の定義で提供しておいて、後からテンプレートにするということもよく起こります。この時でも、使う側は全く変更無しですませることができます。

いままで作ってきた Range クラスのヘッダは次のようなものでした。

```
#ifndef RANGE_H_INCLUDED
#define RANGE_H_INCLUDED
#include <iostream>
namespace mylibrary
{

class Range
{
friend std::ostream&
operator<<(std::ostream& os, const Range& range);
friend std::istream&
operator>>(std::istream& is, Range& range);

public:
        Range();
        Range(double lower, double upper);
        double lower() const { return m_lower; }
        double upper() const { return m_upper; }
        bool is_inside(double x) const;
private:
        double m_lower;
        double m_upper;
};

std::ostream&
operator<<(std::ostream& os, const Range& range);
std::istream&
operator>>(std::istream& is, Range& range);
} // namespace mylibrary
#endif // RANGE_H_INCLUDED
```

この部分をテンプレート化してみましょう。この時、Range という名前は、いままでのまま使いたいので、テンプレートで作るクラス名を Basic_Range とし、後で Range クラスをこのテンプレートを使ったものに置き換えます。

プログラム 12.1: テンプレートクラス

```
1  // range.h
2  #ifndef RANGE_H_INCLUDED
3  #define RANGE_H_INCLUDED
4  #include <iostream>
5  namespace mylibrary
6  {
7
8  template<class T> class Basic_Range
9  {
10 template<class U>
```

```
11  friend std::ostream&
12          operator<<(std::ostream& os, const Basic_Range<U>& range);
13  template<class U>
14  friend std::istream&
15          operator>>(std::istream& is, Basic_Range<U>& range);
16
17  public:
18          Basic_Range() : m_lower(0), m_upper(0) {}
19          Basic_Range(T lower, T upper);
20          T lower() const { return m_lower; }
21          T upper() const { return m_upper; }
22          bool is_inside(T x) const;
23
24  protected:
25          T m_lower;
26          T m_upper;
27  };
28
29  typedef Basic_Range<double> Range;
30
31  template<class T>
32  std::ostream&
33  operator<<(std::ostream& os, const Basic_Range<T>& range);
34
35  template<class T>
36  std::istream&
37  operator>>(std::istream& is, Basic_Range<T>& range);
38
39  ////////
40  template<class T>
41  Basic_Range<T>::Basic_Range(T lower, T upper) :
42          m_lower(lower), m_upper(upper)
43  {
44  }
45
46  template<class T>
47  bool
48  Basic_Range<T>::is_inside(T x) const
49  {
50          if((x < m_lower) || (x > m_upper))
51                  return false;
52          return true;
53  }
54
55  template<class T>
56  std::ostream&
57  operator<<(std::ostream& os, const Basic_Range<T>& range)
58  {
59          os << '[' << range.m_lower << ',' << range.m_upper << ']';
60          return os;
61  }
62
63  template<class T>
64  std::istream&
65  operator>>(std::istream& is, Basic_Range<T>& range)
66  {
67          char c;
68          T xmin;
```

```
69          T xmax;
70          is >> c;
71          if (c != '[')
72                  return is;
73          is >> xmin;
74          is >> c;
75          if (c != ',')
76                  return is;
77          is >> xmax;
78          is >> c;
79          if (c != ']')
80                  return is;
81          range.m_xmin = xmin;
82          range.m_xmax = xmax;
83          return is;
84  }
85  }       // namespace mylibrary
86  #endif  // RANGE_H_INCLUDED
```

プログラム 12.1 にはテンプレートによるメンバー関数の定義なども含まれています。

第13章　標準ライブラリ

C++ では処理系が標準的に持つべき様々なライブラリを定めています。ここでは、その中でも特に使用頻度が高いと思われる文字列ライブラリと入出力ライブラリを紹介します。

標準ライブラリの多くはテンプレートの形で提供されています。この形で提供される標準ライブラリを標準テンプレートライブラリ（Starndard Template Library）、あるいは略して STL と呼びます。これから紹介するライブラリはすべてこの標準テンプレートライブラリに属するものです。なお、これらはすべて名前空間 `std` に用意されています。

クラスを利用するにはリファレンスマニュアルが不可欠です。C++の規格書にはもちろん記載されていますが、C++の教科書などにも掲載されているものもありますしライブラリに特化した教科書もあります。また、インターネットの Web ページとして用意されているものもあります。例えば本書の執筆時点（2011 年 2 月）で、

- `http://www.cplusplus.com/reference/`
- `http://www.cppreference.com/wiki/start`

などがあります。

13.1　標準ライブラリの分類

標準ライブラリは機能によって表 13.1 のように分類されています。

13.2　文字列クラスライブラリ

標準ライブラリで用意されている文字列クラスは `string` クラスです。このクラスは、文字列クラステンプレート `basic_string` を使って `char` 型について具体化したクラスです。つまり `basic_string<char>` に対し `typedef` により別名をつけたものです。ヘッダは `<string>` です。

13.2.1　C スタイル文字列と文字配列

前に述べた文字列、すなわち文字型配列に終端文字 0 を付加したものとは全く異なるものです。文字型配列に終端文字 0 を付加した文字列を特に明確にする場合には C スタイル文字列（C-style strings）とか、あるいは C 文字列（C strings）と呼びます。あるいは規格上では NTBS (null-terminated byte string) と呼びます。

C スタイル文字列と `string` クラスオブジェクトとの大きな違いの一つは、記憶域での連続性です。C スタイル文字列は文字型配列の一つの形と見ることができます。従って、記憶域上では一文字毎に連続した番地に割り当てられています。一方、`string` クラスでは、そのような保証はあり

表 13.1: 標準ライブラリの分類

分野	Category
言語常備	Language support
診断	Diagnostics
一般ユーティリティ	Genral utilities
文字列	Strings
現地化	Localization
コンテナ	Containers
反復子	Iterators
アルゴリズム	Algorithms
数値処理	Numerics
入出力	Input/output

表 13.2: 言語常備関連ヘッダ

ヘッダ	概要	C でのヘッダ
`<limits>`	numerical limits	
`<new>`	dynamic memory allocation	
`<typeinfo>`	runtime type information	
`<exception>`	exception handling	
`<climits>`	C style scalar limits	`<limits.h>`
`<cfloat>`	C style floating point limits	`<float.h>`
`<cstddef>`	`size_t`、`NULL` macro など	`<stddef.h>`
`<cstdarg>`	variable length arguments	`<stdarg.h>`
`<csetjmp>`	setjump longjump	`<setjmp.h>`
`<cstdlib>`	`exit()` `abort()` など	`<stdlib.h>`
`<ctime>`	`clock()` など	`<time.h>`
`<csignal>`	`signal()` など	`<signal.h>`

表 13.3: コンテナ関連ヘッダ

ヘッダ	概要
`<vector>`	one dimensional array
`<list>`	bi-directional list
`<deque>`	double queue
`<stack>`	stack
`<map>`	associated array
`<set>`	set
`<bitset>`	bool array

表 13.4: 入出力関連ヘッダ

概要	ヘッダ	C でのヘッダ
前方宣言	<iosfwd>	
標準ストリームオブジェクト	<iostream>	
入出力ストリーム基底クラス	<ios>	
ストリームバッファ	<streambuf>	
整形と操作子	<istream> <ostream> <iomanip>	
文字列ストリーム	<sstream> <cstdlib>	 <stdlib.h>
ファイルストリーム	<fstream> <cstdio> <cwchar>	 <stdio.h> <wchar.h>

ません。つまりポインタを使って要素の文字を順にたどることはできません（代替手段が提供されています)。

また、string には終端文字という概念はありません。値 0 の文字も文字列の要素として保持することができます。文字列が保持している文字の個数はメンバ関数 length() あるいは size() (両者の内容は同じものです）で得ることができます。なお、この戻り値の型は処理系に依存していて string::size_type と型名が付けられています（符号無し整数型のどれかを typedef したものです)。この型の定数として最後の有効文字の次の位置を表す定数 string::npos があります。

C++ でも使える C の標準ライブラリ関数の中には、C スタイル文字列を引数にするものが多数あります。また外部とのデータのやりとりでは記憶域上での連続性（つまり文字型配列）を要求される場合もあります。string クラスはメンバ関数 c_str、 data 及び copy でこれに対応しています。このうち c_str と data はいずれも記憶域上で連続した領域（すなわち文字型配列）を確保し、保持している文字列を書き出し、確保した配列のポインタを返します。c_str は末尾に値 0 を追加するのに対し、data はこれを行いません。非常に重大な注意として、この配列の管理は呼び出されたオブジェクト側で行うということがあります。つまり、このポインタは元のオブジェクトに変更が加えられた時点で無効になります（実際にいつの時点で無効になるかは別として)。

13.2.2 コンストラクタ

コンストラクタは文字列に関連した様々な型を用いて初期化できます。C スタイルの文字列で初期化することもできます。ただし char を含め数値型では初期化できません。

```
std::string s1 = "A";
```

あるいは

```
std::string s1("A");
```

は、C スタイルの文字列 "A" で初期化した例です。この場合、s1 は一つの文字 'A' を要素とする string 型オブジェクトです。つまり s1.length() は 1 です。

一方

```
std::string s2 = 'A';
```

はエラーになります。char 型で初期化することはできません。しかし、個数を指定することで、char 型を使った初期化はできます。

```
std::string s3(1, 'A');
```

は、一個の文字 'A' を持つ string 型オブジェクトが生成されます。

13.2.3 代入

当然ながら string 型オブジェクトは代入ができます。

```
std::string s1 = "ABC";
std::string s2 = "AIUEO";
s1 = s2;
```

とすると、最後の代入文により s1 の値は "AIUEO" になります。

代入できるのは string 型のオブジェクトだけでなく、C スタイル文字列も代入できます。

```
std::string s1 = "ABC";
std::string s2 = "AIUEO";
s1 = s2;
s2 = "abc";
```

とすると、最後の代入文が終了した時点で、s1 の値は "AIUEO" に、s2 の値は "abc" になります。

紛らわしいことに char での初期化はできませんが、代入は許されています。つまり

```
std::string s1 = 'A';
```

はエラー（char 型による初期化）ですが、

```
std::string s1;
s1 = 'A';
```

は許されます。

13.2.4 要素へのアクセス

string 型オブジェクトの個々の要素は添字を [] で囲むことでアクセスできます。配列と同じように添字は 0 から始まります。添字の型は string::size_type です。

```
std::string s("ABCdefg");
```

とした時、s[0] の値は文字 'A' です。また s[5] の値は文字 'f' です。有効な添字の範囲は 0 から s.length()-1 までです。範囲のチェックは行われません。

範囲のチェックを行いながら添字でアクセスするにはメンバ関数 at を使います。上の例では s.at(2) とするとその値は文字 'C' になります。範囲外を指定すると例外 out_of_range が投げられます。例えば今の場合 s.at(7) とすると例外 out_of_range が投げられますが s[7] とした場合は何が起こるか予測できません。

13.2.5 比較

比較は数値比較で用いられる比較演算子（表 2.1）を用いる方法とメンバ関数 compare を用いる方法があります。大小関係は、辞書順で前の方が小さい値とされます。比較は string 型オブジェクトだけでなく、C スタイル文字列との比較も可能です。

メンバ関数 compare には引数の個数や型の異なるものがいくつかありますが、どれも戻り値は int で、同じ値なら 0、引数として与えられた文字列よりも辞書的に前ならば負の整数値、逆に後ならば正の整数値です。メンバ関数 compare のうち、最も単純なものは引数が一つで、string 型オブジェクトか C スタイル文字列を与えます。

```
std::string s = "ABCdefg";
int n = s.compare("AB");
```

とすると、"ABCdefg" は "AB" より辞書的に後ですので、n の値は正の整数値になります。

開始位置と文字数を指定して比較することもできます。

```
std::string s = "ABCdefg";
int n = s.compare(1, 2, "BC");
```

とすると、s の添字 1 から開始して 2 文字、つまり s のうちの"BC" の部分と比較することになりますので、n の値は 0（一致）となります。

13.2.6 追加

生成した string 型オブジェクトに対し、演算子 += を使って末尾に文字や文字列を追加していくことができます。追加できるのは、string 型オブジェクトの他、C スタイル文字列と char 型文字です。

```
std::string s = "ABC";
s += "def";
s += 'g';
```

とすると、最終的に s の値は "ABCdefg" になります。なお、char 型文字の追加はメンバ関数 push_back を使って行うこともできます。つまり、上の最後の行は

```
s.push_back('g');
```

と書いても結果は同じです。

13.2.7 連結

演算子 + を使って二つの string オブジェクトを連結した string オブジェクトを作ることができます。この場合、一方の項は string 型オブジェクトだけでなく、C 型文字列や文字でも構いません。ただし、どちらか一方は string オブジェクトである必要があります。

```
std::string s;
std::string h = "Hello, ";
s = h + "world";
```

は正しいプログラムですが、

```
std::string s;
s = "Hello, " + "world";
```

はエラーです。二項演算子 + の両側が両方とも C スタイル文字列だからです。

13.2.8 探索

string クラスは文字または文字列を探すメンバ関数をいくつか持っています。ここでは、find と rfind について簡単に述べます。これらは引数として与えられた文字または文字列を探します。前者は先頭から末尾へ向かって探すのに対し、後者は末尾から先頭に向かって探します。戻り値は探している文字または文字列が始まる先頭の添え字の値です。例えば

```
std::string s = "hello, world";
std::string::size_type n = s.find("wor");
```

とすると、n の値は 7 になります。一致する箇所が一箇所の場合は find も rfind も同じ値を返します。見つからなかった場合は string::npos が返ります。

探索の開始位置を指定する場合は、引数を二つ与え、第 1 引数には探索する文字または文字列を与え、第 2 引数には string::size_type 型で開始位置を指定します。

13.2.9 置換と消去

置き換えは replace、消去は erase メンバ関数を使います。置き換えの場合は、開始位置と置き換える個数、置き換える文字列を指定します。更に置き換える文字の開始位置と個数を指定することもできます。動作的には、まず開始位置から置き換える個数分の文字が消去され、次にその位置に置き換える文字が挿入されます。例えば

```
std::string s = "hello, C++ world";
s.replace(7, 3, "Java");
```

とすると、結果の s は "hello, Java world" となります。

消去は開始位置（省略時は 0）と終端位置（省略時は string::npos）を指定します。両方省略すると、全文字が消去されます。

13.2.10 挿入

挿入は insert メンバ関数を使います。最も単純には挿入する位置と文字列を指定するだけですが、挿入する文字列の何文字目から何文字という指定も可能です。いずれも範囲外の位置やサイズを指定すると（終端への挿入は許されます）例外が投げられます。単純な例を示します。

```
std::string s = "hello, world";
s.insert(7, "C++ ");
```

とすると、結果の s は "hello, C++ world" となります。

表 13.5: 非整形入力メンバ関数

メンバ関数

`int get();`
`istream& get(char& c);`
`istream& get(char* s, streamsize n);`
`istream& get(char* s, streamsize n, char delim);`
`istream& get(streambuf& sb);`
`istream& get(streambuf& sb, char delim);`

`istream& getline(char* s, streamsize n);` `istream& getline(char* s, streamsize n, char delim);`
`istream& readsome(char* s, streamsize n);`
`istream& read(char* s, streamsize n);`

表 13.6: 非整形出力メンバ関数

メンバ関数
`ostream& put(char& c);` `ostream& write(const char* s, streamsize n);`
`istream& flush();`

13.3 入出力ライブラリ

C++ の標準クラスライブラリの中に入出力を扱うクラスとして、入出力ストリームクラスが用意されています。これもクラステンプレートとして提供されているものですが、char 型に対しては typedef により通常のクラスとして扱える名前が与えられています。ここではこの char 型のクラスとして述べます。

このクラスは名前の通り入出力をストリーム (stream)、つまり一連のデータの流れとして扱います。このクラスにはデータの流れの向きに応じて、3 種類のクラスが用意されています。

- istream – 入力を扱うクラスです。データはクラスオブジェクトを通じてアプリケーションプログラムへ読み込まれます。
- ostream – 出力を扱うクラスです。データはアプリケーションプログラムからクラスオブジェクトを通じて書き込まれます。
- iostream – 一つのストリームで入出力両方を扱うクラスです。データの流れは双方向です。

13.3.1 非整形入出力

入出力クラスは演算子 << や >> と操作子との組み合わせによって整形入出力できることはこれまで述べてきました。一方、整形せず未加工のままデータを入出力したい場合もあります。これを行うため、非整形入力に関しては表 13.5 に示すメンバ関数が、非整形出力に関しては表 13.6 に示すメンバ関数が用意されています。

13.3.2 ファイルストリーム

標準入出力だけでも、 redirect を使うと入出力をファイルに切り替えることができますから、一つのファイルから読んで、一つのファイルに書き出すだけであれば、標準入出力だけで十分なことが多々あります。

一方、複数の入力ファイルや出力ファイルを扱う場合には、直接プログラムからファイルの読み書きをする必要が出てきます。このため、標準ライブラリにはファイルストリームと呼ばれるクラスが用意されています。ヘッダ `fstream` をインクルードすることで利用可能になります。

このクラスは入出力ストリームから派生しているので、入出力ストリームクラスの持つ `public` メンバはすべて利用可能です。それに加え、ファイルクラス特有のいくつかの関数を持っています。

オープンとクローズ

ファイルは名前によりオープン（open）することで利用可能になり、クローズ（close）することで利用の終了となります。ファイルストリームクラスでは、オープンに対し二つの方法があります。一つは、引数を持つコンストラクタを利用してオープンする方法、もう一つはデフォルトコンストラクタでオブジェクトを生成した後でメンバ関数 `open` を利用する方法です。

クローズも同様で、デストラクタで自動的に `close` されますが、メンバ関数 `close` を直接呼び出してクローズすることも可能です。

ファイル入出力に関しては プログラム 14.4 やプログラム 14.11 に使用例が示してあります。

13.3.3 文字列ストリーム

文字列を扱う `string` クラスには入出力ストリームが提供するような入出力整形機能はありません。入出力ストリームから派生させた `stringstream` クラスは文字列から整形入力を行ったり、文字列へ整形出力を行ったりする機能を提供します。基底クラス `istream` からは `istringstream`、`ostream` からは `ostringstream`、`iostream` からは `stringstream` クラスがそれぞれ派生しています。ヘッダは `<sstream>` です。

この使用例がプログラム 14.6 に示してあります。

第14章　応用プログラムを作る

プログラム言語を知っているからと言ってプログラムが書けるわけではありません。自動車の運転の仕方を知っているからと言って、実際に自動車が運転できるわけではないのと一緒です。プログラムが書けるようになるためには、実際に使えるプログラムを書いてみることが重要です。そこで、ここから、いくつかの応用プログラムを作ってみます。

14.1　テキストファイルを Web ページに変換

テキストファイルというのは、文字や改行や空白などのいくつかの制御文字に対し、そのコードをそのままファイルにしたものです。フォントや文字強調や位置決めなどの情報を含まないファイルです。Windows で言えば、メモ帳で作るファイルです。C や C++ のソースファイルも、テキストファイルです。

Web ページの基本である `html` ファイルもテキストファイルですが、いくつかの文字は Web ページの表示の制御などに使われます。従って、通常のテキストファイルを Web ページに埋め込む場合には、これらの文字を変換する必要があります。具体的には、次の変換をする必要があります。

```
<       →       &lt;
>       →       &gt;
&       →       &amp;
```

また、Web ページの中で、このファイルを埋め込む部分は何も整形しないという指示が必要で、最初に

```
<pre>
```

という文字列を、最後には

```
</pre>
```

という文字列を付加します。

変換する 3 つの文字を判定するのに、`if` 文を使うプログラム 14.1 と `switch` 文を使うプログラム 14.2 を示します。

これらのプログラムは、既存の Web ページの中へ埋め込むことを想定していますが、単独の Web ページにしたい場合は、前後にもう少し加える必要があります。必要最低限のものを加えた例をプログラム 14.3 に示しておきます。この他に文字コードの指定などが必要になる場合があります。

プログラム 14.1: Web ページへの変換 (1)

```
#include <iostream>
int main()
{
        std::cout << "<pre>" << std::endl;
        char c;
        while (std::cin.get(c))
                if (c == '<')
                        std::cout << "&lt;";
                else if (c == '>')
                        std::cout << "&gt;";
                else if (c == '&')
                        std::cout << "&amp;";
                else
                        std::cout.put(c);
        std::cout << "</pre>" << std::endl;
        return 0;
}
```

プログラム 14.2: Web ページへの変換 (2)

```
#include <iostream>
int main()
{
        std::cout << "<pre>" << std::endl;
        char c;
        while (std::cin.get(c))
                switch (c)
                {
                case '<':
                        std::cout << "&lt;";
                        break;
                case '>':
                        std::cout << "&gt;";
                        break;
                case '&':
                        std::cout << "&amp;";
                        break;
                default:
                        std::cout.put(c);
                        break;
                }
        std::cout << "</pre>" << std::endl;
        return 0;
}
```

プログラム 14.3: Web ページへの変換 (3)

```
1  #include <iostream>
2  int main()
3  {
4          std::cout
5                  << "<html>\n"
6                  << "<head><title>Plain text</title></head>\n"
7                  << "<body>\n";
8
9          std::cout << "<pre>" << std::endl;
10         char c;
11         while (std::cin.get(c))
12                 switch (c)
13                 {
14                 case '<':
15                         std::cout << "&lt;";
16                         break;
17                 case '>':
18                         std::cout << "&gt;";
19                         break;
20                 case '&':
21                         std::cout << "&amp;";
22                         break;
23                 default:
24                         std::cout.put(c);
25                         break;
26                 }
27         std::cout << "</pre>\n";
28
29         std::cout
30                 << "</body>\n"
31                 << "</html>"
32                 << std::endl;
33         return 0;
34 }
```

14.2 16進ファイルダンププログラム

ここでファイルを16進数で表示するプログラムを考えてみます。データなどを格納してあるファイルがうまく読めない時など、ファイル構造がどうなっているのか調べるのに便利です。この場合、ファイルはテキストファイルだけでなくデータをバイナリで記録したものも調べられるようにしましょう。そのためには、ファイルをバイナリモードでオープンする必要があります。つまり、標準入出力のように既にファイルが開いている状態の入力ストリームを扱うのではなく、プログラムの中からファイルを開く必要があります。そのためにはファイルの名前が必要です。まずはこの部分から考えましょう。

14.2.1 main関数のもう一つの形

いままで、main関数は引数無しとして扱ってきました。実はmain関数には

```
int main(int argc, char* argv[])
```

という引数を二つとる形式のものがあります。これらの引数は実行環境により与えられ、第二引数であるchar* argv[]はC型文字列へのポインタの配列で、第一引数のint argcは、その配列の個数です。

どのような値が渡されるかは一般には実行環境に依存しますが、多くの場合、以下に述べるようになっています。例えば今プログラムhexdumpを実行するため、

```
hexdump -N 100 myfile.dat
```

とキーボードから指示したとしましょう。この場合は

```
argc = 4;
argv[0] = "hexdump";
argv[1] = "-N";
argv[2] = "100";
argv[3] = "myfile.dat";
```

として値が渡ってきます。このように実行指令の文字列を空白部分で区切って渡してくるわけです。

14.2.2 プログラムの外からファイル名を与える

ファイル名をプログラムの外から与えるために、14.2.1の仕掛けを使いましょう。作成したプログラムをプログラム14.4に示します。このファイルをhexdump.cppとして、実行ファイルhexdump（Windowsの場合はhexdump.exe）を作り、

```
hexdump hexdump.cpp
```

とすると

プログラム 14.4: 16 進ダンプ第 1 版

```
1  #include <iostream>
2  #include <fstream>
3  #include <iomanip>
4
5  int main(int argc, char* argv[])
6  {
7          if (argc != 2)
8                  return 0;
9
10         int nout = 0;
11         std::cout << std::hex << std::setfill('0');
12
13         std::ifstream ifs(argv[1], std::ios::binary);
14         char c;
15         while (ifs.get(c))
16         {
17                 std::cout << ' ' << std::setw(2)
18                         << (static_cast<int>(c) & 255);
19                 if ((++nout) >= 16)
20                 {
21                         std::cout << '\n';
22                         nout = 0;
23                 }
24         }
25         std::cout << std::endl;
26         return 0;
27 }
```

```
23 69 6e 63 6c 75 64 65 20 3c 69 6f 73 74 72 65
61 6d 3e 0d 0a 23 69 6e 63 6c 75 64 65 20 3c 66
73 74 72 65 61 6d 3e 0d 0a 23 69 6e 63 6c 75 64
65 20 3c 69 6f 6d 61 6e 69 70 3e 0d 0a 0d 0a 69
6e 74 20 6d 61 69 6e 28 69 6e 74 20 61 72 67 63
2c 20 63 68 61 72 2a 20 61 72 67 76 5b 5d 29 0d
:
```

のような結果が得られるでしょう。

プログラムの概要

さて、この プログラム 14.4 を眺めてみます。14.2.1 に述べたように、ここでは二つの引数をとる main 関数を使っています。文字列の個数は二つ、最初はプログラム名で二番目がファイル名であることを期待しているので、7 行目の if 文で渡された文字列が二つでない場合は、すぐに終了しています。使いやすくするには、この点は少し工夫が必要ですが、ここでは簡単のためこうしています。

10 行目と 11 行目は出力の準備を行っています。見やすくするために 1 行につき 16 バイトを表示するために、nout という変数を用意し、何バイト処理したかを数えます。また標準出力の整形モードを 16 進表示に、桁の無い部分は '0' で埋めるように設定しています。

13 行目で、入力ファイルはコンストラクタで開いています。ファイル名は main 関数の第二引数の二番目の文字列、つまり argv[1] で与えられるとしています。この点についても改良が必要ですが、後で考察することにします。

与えられたファイル名のファイルが存在しない場合などはどうなるのでしょう？標準入出力ライブラリは、特に設定をしない限り例外を投げません。代わりに状態を持ち、正常であれば good、要求された操作が実行できない場合は fail や bad になります。またオブジェクトの参照で good であるか否かを調べることができるように演算子 ! や void* が上書きされています。15 行目は、これを使って 1 バイトずつ読みだしています。読みだせなかった場合は while 文の条件部が false になり、繰り返しは終了します。例えばファイルが存在しない場合は最初から読みだせないので何も出力せずに終了します。

繰り返し文の中は、単に読みだした文字を 16 進数として表示するためのものです。若干複雑に見えるのは、16 バイト処理する毎に改行を入れる処理が入っているためです。操作子の setw は、数値オブジェクトを処理する毎にリセットされるので、毎回設定し直しています。

文字 c をわざわざ int に変換しているのは、整形出力で文字を整数として表示させたいからです。更に符号拡張がおこっても対処できるよう、255 とビット単位の論理積をとって下位 8 ビットのみが有効になるようにしています。

14.2.3 オプションを処理する

プログラム 14.4 は使う側からすると非常に使いにくいプログラムです。まず第一にファイル名を与えない場合、あるいは存在しないファイル名を与えた場合、何も表示せずに終了します。ファイル名を二つ与えても、あるいはファイル名の他に何か文字列を入れても何も表示せずに終了しま

す。第二には、一部だけ表示することができません。大きなファイルの終わり付近だけを見たくても全部表示させる必要があります。第三に何バイト目であるかが出力されないので、少し大きくなると何バイト目なのかを知るのが非常に困難になります。

ここでは、これらの問題の解決を試みます。

オプションの与え方

コマンドラインを使ってプログラムを実行させる方式では、コマンドラインにオプションを与えて実行させることにより様々な要求に応えられるようにする方法がよく採られます。ここでも、この方式を採用しましょう。オプションとして、

- 何バイト目から表示するか
- 何バイト分表示するか

が与えられるようにしましょう。この与え方は自由に決めることができますが、ここでは Linux における同様なプログラム od に合わせて、「何バイト目から」については -j の次の数値で、「何バイト分表示するか」については-N の次の数値で与えるようにします。例えば、mydata.dat というファイルの 100 バイト目から 50 バイト分表示するには

```
hexdump -j 100 -N 50 mydata.dat
```

のように与えます。

オプションを保持するクラスを作る

オプションは人間が与えますから、誤りもあるでしょう。また順番も自由に与えられた方がよいでしょう。そこでオプションを保持するクラスを作ってしまいましょう。これらの処理はプログラム本体とは別の処理ですので、クラスにしてしまいましょう。つまり、このクラスオブジェクトは、与えられたコマンドラインの文字列を解析してプログラム本体に必要なファイル名とオプションを保持するようにしましょう。

本体であるプログラム 14.6 では、引数を持つコンストラクタを定義しています。この時の引数は main 関数の引数をそのまま使うことを想定しています。従って、プログラム名やファイル名の文字列もポインタだけをコピーしています。また、表示バイト数の初期値は 0 としていますが、この値 0 は hexdump の方で無制限として扱います。

文字列から unsigned long の値を得るのに、std::stringstream クラスを使っています。これは入出力ストリームの派生クラスとして用意されている標準ライブラリの一つで、標準入出力ストリームと同様、整形入出力が行えます。また、オブジェクトを bool に型変換できて、処理がうまく行えなかったときは false になるので、オブジェクトそのものを if 文などの条件部に使うことができます。

予期せぬ型変換

ヘッダをよく見ると、型変換演算子 void* と論理否定演算子 ! が定義されています。これは標準入出力ストリームなどで行われているように、オブジェクトそのもので状態を調べられるように

プログラム 14.5: DumpOpt クラスのヘッダ

```
1  #ifndef DUMPOPT_H_INCLUDED
2  #define DUMPOPT_H_INCLUDED
3  #include <ostream>
4  class DumpOpt
5  {
6  public:
7          DumpOpt(int argc, char* argv[]);
8          const char* progname() { return m_progname; }
9          const char* filename() { return m_filename; }
10         unsigned long offset() { return m_offset; }
11         unsigned long outbytes() { return m_outbytes; }
12         operator void* () const
13                 { return (m_status == GOOD) ? (void*)this : 0; }
14         bool operator!() const { return (m_status != GOOD); }
15         int usage(std::ostream& os);
16
17 public:
18         enum Status
19         { GOOD, NOFILE, MULTIFILE, INVALIDKEY, INVALIDNUM, NONUM };
20
21 private:
22         Status m_status;
23         const char* m_progname;
24         const char* m_filename;
25         unsigned long m_offset;
26         unsigned long m_outbytes;
27 };
28 #endif  // DUMPOPT_H_INCLUDED
```

プログラム 14.6: DumpOpt クラスの本体

```
1  #include "dumpopt.h"
2  #include <sstream>
3  DumpOpt::DumpOpt(int argc, char* argv[]) :
4          m_status(NOFILE), m_progname(0), m_filename(0),
5          m_offset(0), m_outbytes(0)
6  {
7          if (argc < 1)
8                  return;
9          m_progname = argv[0];
10         int i = 1;
11         while (i < argc)
12         {
13                 const char* s = argv[i++];
14                 if (*s != '-')
15                 {
16                         if (m_filename != 0)
17                         {
18                                 m_status = MULTIFILE;
19                                 m_filename = 0;
20                                 return;
21                         }
22                         m_filename = s;
23                 }
24                 else
25                 {
26                         char c = *(s + 1);
27                         if ((c == 'N') || (c == 'j'))
28                         {
29                                 if (i >= argc)
30                                 {
31                                         m_status = NONUM;
32                                         return;
33                                 }
34                                 std::stringstream ss(argv[i++]);
35                                 unsigned long lv;
36                                 if (!(ss >> lv))
37                                 {
38                                         m_status = INVALIDNUM;
39                                         return;
40                                 }
41                                 if (c == 'N')
42                                         m_outbytes = lv;
43                                 else if (c == 'j')
44                                         m_offset = lv;
45                         }
46                         else
47                         {
48                                 m_status = INVALIDKEY;
49                                 return;
50                         }
51                 }
52         }
53         if (m_filename)
54                 m_status = GOOD;
55 }
```

プログラム 14.7: 予期せぬ型変換

```
1  #include <iostream>
2  class MyInt
3  {
4          int m_value;
5  public:
6          MyInt(int n) : m_value(n) {}
7          operator bool () const { return m_value ? true : false; }
8  };
9
10 int main()
11 {
12         MyInt mi = 5;
13         if (mi)
14                 std::cout << "True\n";
15         else
16                 std::cout << "False\n";
17         std::cout << mi + 2 << std::endl;
18         return 0;
19 }
```

するためです。では演算子 ! は当然として、なぜ型変換演算子 bool を用意するのではなく型変換演算子 void* なのでしょう？これは、型変換の規則、特に整数型の昇格がその理由です。処理系は int 値が必要な場合には、当然 int 型への変換を試みます。もし、そのクラスに int 型への変換は定義されていないけれども、bool 型への変換が定義されている場合、bool 型は int 型に昇格できるので、これが使われてしまいます。そうすると予期せぬ結果が起こります。

簡単な例で調べてみましょう。プログラム 14.7 は単に整数を保持するだけのクラスで、コンストラクタの他には bool への変換だけが定義されています。ところが、このプログラムに見るように、このクラスオブジェクトは整数との加算ができてしまって、しかも結果は 3 という値になります。これは、MyInt 型から int 型へ変換を試みて、int への変換が定義されていないためにより範囲の狭い型を調べ bool 型が見つかったので、これが使われ、true が返ってきたので、規則により整数値 1 とされ、それに整数リテラル 2 が加算されたためです。

こうして見ると、小さい範囲しかとれない型に対する変換を与えてしまったことが問題の原因であることがわかります。より大きな範囲の型との演算では、より大きな範囲の型へ必ず変換できてしまいます。この変換を行わせないようにするために、void* への変換を使います。なぜなら、void* は、任意の型のポインタを格納できる、つまり最大の格納サイズのポインタであり、これ以上変換されることはないからです。ポインタは bool へ変換できますから、処理系は bool が必要なところで、他に変換できる候補が無ければ void* 変換を使います。

14.2.4 オプションを保持するクラスを使う

話が横道へそれましたが、オプションを保持するクラスができたので、16 進ファイルダンププログラムを完成させましょう。出力が見やすくなるように 16 バイト処理するごとに改行を入れるのは前と同じですが、ファイルの途中からも表示することがあるので、少し複雑になります。ファイルの位置を 16 進数で表示する部分も加えました。

なお、プログラムの使用法を示す関数 usage をオプション処理側に入れました。オプションを

プログラム 14.8: 16 進ダンプ第 2 版

```
1  #include <iostream>
2  #include <fstream>
3  #include <iomanip>
4  #include "dumpopt.h"
5
6  int main(int argc, char* argv[])
7  {
8          DumpOpt opt(argc, argv);
9          if (!opt)
10                 return opt.usage(std::cerr);
11
12         unsigned long offset = 0;
13         unsigned long outbytes = 0;
14
15         offset = (opt.offset() / 16) * 16;
16
17         int nout = 0;
18         std::cout << std::hex << std::setfill('0');
19
20         std::ifstream ifs(opt.filename(), std::ios::binary);
21         if (!ifs)
22         {
23                 std::cerr << "File␣open␣error:␣"
24                         << opt.filename() << std::endl;
25                 return (-1);
26         }
27         ifs.seekg(offset, std::ios::beg);
28         char c;
29         while (ifs.get(c))
30         {
31                 if (nout == 0)
32                         std::cout << std::setw(8) << offset << ':';
33                 int x = c; x &= 255;
34                 if (offset < opt.offset())
35                         std::cout << "␣␣␣";
36                 else
37                 {
38                         std::cout << '␣' << std::setw(2) << x;
39                         ++outbytes;
40                 }
41                 ++offset;
42                 if ((++nout) >= 16)
43                 {
44                         std::cout << '\n';
45                         nout = 0;
46                 }
47                 if (opt.outbytes() && (outbytes >= opt.outbytes()))
48                         break;
49         }
50         std::cout << std::endl;
51         return 0;
52 }
```

プログラム 14.9: 使用法を定義

```
1 #include "dumpopt.h"
2 int DumpOpt::usage(std::ostream& os)
3 {
4         os << "Usage: " << m_progname << " [options] filename\n";
5         os << "  -j n     start offset\n";
6         os << "  -N n     output bytes" << std::endl;
7         return m_status;
8 }
```

変更する場合など、この方が管理しやすいでしょう。

いくつか使ってみます。まず何も与えずプログラム名だけで実行してみます。

```
C:\Users\cpp>hexdump
Usage: hexdump [options] filename
 -j n    start offset
 -N n    output bytes
```

簡単な使用法が表示されます。

表示するバイト数だけ指定すると、先頭から指定バイト数だけ表示します。

```
C:\Users\cpp>hexdump -N 100 hexdump.exe
00000000: 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
00000010: b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00
00000040: 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
00000050: 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
00000060: 74 20 62 65
```

途中から表示することもできます。100 バイト目から（つまり上の続きから）100 バイト表示してみます。

```
C:\Users\cpp>hexdump -N 100 -j 100 hexdump.exe
00000060:             20 72 75 6e 20 69 6e 20 44 4f 53 20
00000070: 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
00000080: 50 45 00 00 4c 01 07 00 1d cf 3f 4d 00 c2 04 00
00000090: 4e 18 00 00 e0 00 07 03 0b 01 02 38 00 7c 04 00
000000a0: 00 be 04 00 00 58 00 00 60 11 00 00 00 10 00 00
000000b0: 00 90 04 00 00 00 40 00 00 10 00 00 00 02 00 00
000000c0: 04 00 00 00 01 00 00 00
```

存在しないファイルを指定するとどうなるでしょう？

```
C:\Users\cpp>hexdump hhexdump.exe
File open error: hhexdump.exe
```

ファイルが開けないと言ってきます。

これで、とりあえず使えるレベルにまではなりました。しかしまだ問題は残っています。例えばファイルサイズを超えて開始位置を指定すると何も言わずに終わるでしょう。また、オプションの指定に誤りがあった場合、常に usage を示しているだけですが、どういう誤りなのか（例えば DumpOpt の m_status に応じて）メッセージを出す方がよいでしょう。挑戦してみてください。

表 14.1: AU ファイル形式ヘッダ部

位置 (32 ビット単位)	意味	内容
0	magic number	ASCII 文字列で ".snd"、16 進数で 0x2e736e64。
1	data offset	8 ビット単位でのデータ開始位置。少なくとも 24。
2	data size	8 ビット単位でのデータサイズ。未知の時は、0xffffffff。
3	encoding	エンコード方式。Linear PCM 8bit は 2、16bit は 3 など。
4	sample rate	毎秒あたりのサンプル数
5	channels	音源のチャネル数、1 は mono、2 は stereo など。

14.3 音のファイルを作ってみる

データをコンピュータファイルに記録することはデータ収集の基本です。ここでは、コンピュータ上でデータが簡単に生成できて、確認も比較的容易な音のデータを扱ってみます。

音は、空気の振動です。音のコンピュータファイルというのは、その振動の大きさ（振幅）を音の振動周期よりも短い周期で計測しデジタル記録したものです。一般に連続的に変化するアナログ信号を非常に短い間隔で計測しデジタル符号化してデータ列として表現する方法をパルス符号変調（PCM; pulse code modulation）と呼びます。ここで扱う音のデジタル記録方式は、その典型的な例です。

再生プログラムは、その記録を読み出して、スピーカやイヤフォンなど音を出す機器に同じ振動を与えて空気を振動させ、人間の耳に同じ音として認識させるわけです。

音のファイルには、様々な形式があります。その理由の一つは計測した周波数や、ステレオで記録したものか、モノラルなのかなどの属性情報を格納する方式が色々あること、もう一つはファイルサイズが大きくなるために通常は圧縮技術が使われ、その圧縮法が色々あるためです。ここでは、非常に単純で多くの再生プログラムが対応している音のファイル形式として Sun Microsystems によって導入された AU ファイル形式のファイルを作ってみます。このファイル形式のヘッダ部を表 14.1 に示します。ここで 32 ビット word は big endian（E 参照）です。実際の PCM データは、ファイル上で data offset から始まります。

同様に単純に音を格納できるファイル形式としては、マイクロソフトの WAV ファイルがあります。現在では、こちらの方が対応する再生ソフトウェアが多いと思われますが、AU ファイル形式の方がより単純な構造なので、ここでは AU ファイルの方を扱います。

14.3.1 音のファイルクラスの考察

クラスを考える時に最も重要なのはコンストラクタであると筆者は考えています。よく設計されたコンストラクタを持つクラスは非常に使いやすくなります。資源の確保と必要なパラメータの正しい初期化がコンストラクタの役割です。クラスオブジェクトとして生成されたならば、そのオブジェクトは直ちに使えることが基本です。

もちろん、これまで何度も述べているように C++ 言語はプログラミングスタイルを強制しません。従って、従来の C プログラミングスタイルのように、コンストラクタは単に記憶域を確保するだけで、パラメータの初期化は初期化関数を呼び出すというスタイルで記述することも可能で

す。しかし、それではクラス化する利点の多くを失っています。クラスとして設計するからには、使用の際にコンストラクタの他に初期化関数を呼び出す必要があるような設計は根本的に間違っています。なぜなら、この方式では、正しく初期化されないまま使用される可能性が排除できないからです。もちろん多数のパラメータがあって、コンストラクタだけではすべてを指定するのは現実的でない場合もあるでしょう。そのような場合、パラメータを設定し直す関数を用意することは一つの解決策を与えます。しかし、その場合であっても、その関数を使うことなく使用したとしても正しく使えるべきです。

このことを念頭に置いて、AU ファイルを考察してみましょう。

コンストラクタの引数

まずパラメータですが、ファイル仕様を見ればわかるように、わずか 6 つです。このうち、magic number は AU ファイルであることを示す固定のデータですので、ユーザが設定すべきものではありませんし変数ですらありません。更に data offset と data size もプログラム自身が設定すべきもので、ユーザが設定すべきものではありません。つまり、ユーザが設定すべきパラメータは encoding、sample rate 及び channels の 3 つです。この他にファイルの名前が必要ですので、ファイル名もコンストラクタで与えるべきでしょう。

省略時の値の考察

ここまでの考察により、コンストラクタが持つ引数は

- ファイル名
- encoding
- sample rate
- channels

と決めました。少ないとはいえ、使う立場からすると、とりあえず使ってみようという時に、これら全部を設定するのは面倒です。省略可能なものは適宜省略時の値を設定することにして、最小限の設定で使えるようにしましょう。

ファイル名は省略できないことにします。つまり、引数無しのコンストラクタは作らないことにします。次の encoding は、実装は linear PCM、つまり音の信号の最大振幅が整数の最大値になるように線形に比例させる方式のみとしましょう。実装する整数は、8、16、24 および 32 ビットの整数としましょう。

14.3.2 テストプログラム

さて、AUofstream の実装ができたので、いくつかテストプログラムを作って動かしてみましょう。

プログラム 14.10: 音のファイルクラス（ヘッダ）

```
1  // auofstream.h
2  #ifndef AUOFSTREAM_H_INCLUDED
3  #define AUOFSTREAM_H_INCLUDED
4  #include <fstream>
5  namespace mylibrary
6  {
7  class AUofstream : std::ofstream
8  {
9  public:
10         typedef unsigned int     UINT32;
11         typedef signed int       SINT32;
12         typedef unsigned short   UINT16;
13         typedef signed short     SINT16;
14         typedef unsigned char    UINT8;
15         typedef signed char      SINT8;
16         enum ENCODING
17         {
18                 LINEAR_PCM_8BIT = 2,
19                 LINEAR_PCM_16BIT,
20                 LINEAR_PCM_24BIT,
21                 LINEAR_PCM_32BIT
22         };
23         enum CHANNELS
24         {
25                 MONAURAL = 1,
26                 BINAURAL
27         };
28 public:
29         AUofstream(const char* filename,
30                 UINT32 samplerate = 44100,
31                 CHANNELS channels = BINAURAL,
32                 ENCODING encoding = LINEAR_PCM_16BIT);
33         bool putdata(double height);
34         bool putdata(double left, double right);
35         bool fail() const { return std::ofstream::fail(); }
36         UINT32 samplerate() const { return m_samplerate; }
37
38 private:
39         UINT32 m_dataoffset;
40         UINT32 m_datasize;
41         UINT32 m_encoding;
42         UINT32 m_samplerate;
43         UINT32 m_channels;
44         static const char m_magic[4];
45
46 private:
47         bool putUINT32(UINT32 uval);
48         bool put(double dat);
49         bool put(long int ival);
50 };
51 }       // namespace mylibrary
52 #endif  // AUOFSTREAM_H_INCLUDED
```

プログラム 14.11: 音のファイルクラス (1/2)

```
1  // auofstream.cpp
2  #include "auofstream.h"
3  namespace mylibrary
4  {
5  const char AUofstream::m_magic[] = {0x2e, 0x73, 0x6e, 0x64};
6
7  AUofstream::AUofstream(const char* filename,
8          UINT32 samplerate, CHANNELS channels, ENCODING encoding) :
9          std::ofstream(filename, std::ios::binary),
10         m_dataoffset(24),
11         m_datasize(0xffffffff),
12         m_encoding(encoding),
13         m_samplerate(samplerate),
14         m_channels(channels)
15 {
16         if (!std::ofstream::write(m_magic, 4))
17                 return;
18         if (!putUINT32(m_dataoffset))
19                 return;
20         if (!putUINT32(m_datasize))
21                 return;
22         if (!putUINT32(m_encoding))
23                 return;
24         if (!putUINT32(m_samplerate))
25                 return;
26         if (!putUINT32(m_channels))
27                 return;
28 }
29
30 bool AUofstream::putUINT32(UINT32 uval)
31 {
32         char buf[4];
33         buf[0] = static_cast<char>((uval >> 24) & 255);
34         buf[1] = static_cast<char>((uval >> 16) & 255);
35         buf[2] = static_cast<char>((uval >> 8)  & 255);
36         buf[3] = static_cast<char>(uval & 255);
37         if (!std::ofstream::write(buf, 4))
38                 return false;
39         return true;
40 }
41
42 bool AUofstream::putdata(double height)
43 {
44         if (!put(height))
45                 return false;
46         if (m_channels == MONAURAL)
47                 return true;
48         if (!put(height))
49                 return false;
50         return true;
51 }
```

プログラム 14.12: 音のファイルクラス (2/2)

```
1  bool AUofstream::putdata(double left, double right)
2  {
3          if (m_channels == MONAURAL)
4                  return put((left + right) / 2.0);
5          if (!put(left))
6                  return false;
7          if (!put(right))
8                  return false;
9          return true;
10 }
11
12 bool AUofstream::put(double height)
13 {
14         if (height > 1.0)
15                 height = 1.0;
16         else if (height < -1.0)
17                 height = -1.0;
18         long int ival = static_cast<long int>(height * 2147483647.9);
19         if (!put(ival))
20                 return false;
21         return true;
22 }
23
24 bool AUofstream::put(long int ival)
25 {
26         char buf[4];
27         int nout = 0;
28         buf[nout++] = static_cast<char>((ival >> 24) & 255);
29         if (m_encoding != LINEAR_PCM_8BIT)
30         {
31                 buf[nout++] = static_cast<char>((ival >> 16) & 255);
32                 if (m_encoding != LINEAR_PCM_16BIT)
33                 {
34                         buf[nout++] =
35                                 static_cast<char>((ival >> 8)  & 255);
36                         if (m_encoding != LINEAR_PCM_24BIT)
37                                 buf[nout++] =
38                                         static_cast<char>(ival & 255);
39                 }
40         }
41         if (!std::ofstream::write(buf, nout))
42                 return false;
43         return true;
44 }
45 }       // namespace mylibrary
```

プログラム 14.13: 正弦波の音を作る

```
1  #include <iostream>
2  #include "auofstream.h"
3  #include <cmath>
4
5  int main()
6  {
7          mylibrary::AUofstream au("testau01.au");
8          int samplerate = au.samplerate();
9          if (au.fail())
10                 std::cerr << "File creation fail!" << std::endl;
11         else
12                 for (int i = 0; i < samplerate; i++)
13                         au.putdata(sin(
14                                 2.0 * 3.1415926536 * 440.0 * i /
15                                 static_cast<double>(samplerate)));
16         return 0;
17 }
```

正弦波

音のデータを生成するのに、まずは正弦波で作ってみます。音叉の出す音です。時報音としても使われています。音程は周波数で決まります。ここでは 440Hz の音を出してみます。音楽で言えば標準のラの音です。時報で言えば予報音（ピッピッピッ）の音です。因みに時報の正時音（ポーン）は 880Hz です。

音の周波数とパルス符号変調における標本周波数とは別であることに注意します。標本周波数は音の周波数に比べ十分高くなければなりません。ここで作ったクラス AUofstream の省略時周波数は、このことを考慮して決めました。そこで、ここでは何も考えず、この省略時周波数を使いましょう。

音の長さですが、ここでは 1 秒間にしてみます。つまり標本周波数回データを計測して書き込みます。プログラム 14.13 が「1 秒間の正弦波の音」のファイルを作るプログラムです。このプログラムを見てわかるように、著しく簡単です。ファイル名前以外、設定情報は一切書かれていません。このテストプログラムを書く時には、AU ファイルの設定やその構造などはすべて忘れて、「正弦波を標本周波数で計測してその値を書き込む」ということだけに専念できるわけです。それだけで正しく再生できるファイルが出来上がります。ファイルのオープンやクローズすらも呼んでません。クラスを作る時に十分に考察して作ると利用は著しく簡単になります。

もちろん、省略時設定で作られたファイルは効率的ではないことが多々あります。例えば今の例では、省略時はステレオでも使えるように、2 音源の設定になっていて、1 音を与えた時には両方に同じデータを書き込んでます。そのためファイルサイズが必要量の 2 倍になってます。また、この周波数帯の音だけであれば標本周波数は省略時の値である 44100Hz(CD 品質）である必要はなく、いわゆる電話品質（標本周波数 8000Hz）でも十分かも知れません。そうするとファイルサイズは更に 1/5 以下になります。しかし、そのような効率を考える場合ですら、コンストラクタの引数を省略せずに与えるだけで済みます。そのように設計したからです。クラスを作る時には、実装よりも何よりも、どのような使われ方がされるのかを十分に考察して設計することが重要です。利用しやすいクラスにできるか否かは、多くの場合、コンストラクタの設計にかかっています。

プログラム 14.14: うなりを作る

```
1  #include <iostream>
2  #include "auofstream.h"
3  #include <cmath>
4
5  int main()
6  {
7          mylibrary::AUofstream au("testau02.au");
8          int samplerate = au.samplerate();
9          if (au.fail())
10                 std::cerr << "File creation fail!" << std::endl;
11         else
12                 for (int i = 0; i < (samplerate * 5); i++)
13                         au.putdata(
14                                 sin(2.0 * 3.1415926536 * 440.0 * i /
15                                         static_cast<double>(samplerate)),
16                                 sin(2.0 * 3.1415926536 * 441.0 * i /
17                                         static_cast<double>(samplerate)));
18         return 0;
19 }
```

14.3.3 うなりを作ってみる

周波数の僅かに異なる二つの音が同時に出ると、干渉してうなりを生じます。これを実験してみる音のファイルを作ってみましょう。もともと AUofstream クラスは、ステレオ音源に対応できるように設計したので、右側の音と左側の音を同時計測して書き込めばよいだけです。プログラム 14.14 がそれです。このプログラムでは、左側音源は 440Hz、右側音源は 441Hz を与えています。うなりの周波数は、二つの音源の周波数の差ですので、1Hz です。十分繰り返しが聞こえるように 5 秒間のデータにしてあります。このプログラムを実行してできる音のファイル testau02.au をステレオスピーカーを使って再生すると、1Hz のうなりが聞こえるはずです。前のプログラム 14.13 に比べ、ほとんど何も変わっていないことがわかるでしょう。

14.4 C 言語との連携

一般にプログラム言語が異なると、生成されるモジュールの形式が異なります。従って、異なる言語で生成されたモジュールをリンクして一つの実行モジュールを作る際には、異なる形式のモジュールを扱うための手続きが必要になります。

C++ は C 言語の上位互換性を強く意識した言語ですので、C 言語で書かれたソースファイルを、C++ でコンパイルすることにより、この問題を回避することは比較的容易にできます。しかし、過去の資産をすべて C++ でコンパイルし直すことは現実的ではありません。このような場合 C++ には、別言語でコンパイルされたモジュールを使うことを宣言することにより C++ のソースコードの中で別言語でコンパイルされた（される）オブジェクトや関数を使うことができます。

14.4.1 gzip ファイルを扱う

例として、データストリームを圧縮保存する際によく用いられている gzip ファイルを扱ってみます。これは、RFC 1952 に記述されているもので、その実装としては `zlib` と呼ばれている C ライブラリが有名です。このライブラリの中には、gzip ファイルを扱うための関数群が含まれています。この関数群は、C のファイル入出力関数と類似になるように設計されています。まず、この関数群を C++ から使ってみましょう。プログラム 14.15 は圧縮されていないファイルを読んで、gzip 圧縮するプログラムです。いわばユーティリティプログラムである `gzip` の簡易版です。このプログラムを生成するには、`zlib` のライブラリが必要です。GNU compiler collection （gcc や g++ など）で、これを指定するには `-lz` を付けます。つまり、例えば プログラム 14.15 のソースファイルを `gztest.cpp` とすると、

```
g++ gztest.cpp -lz
```

とします。

このプログラムでは、大部分 C の記法で書いています。この中の `fopen()` や `fclose()` は、`<cstdio>` を `#include` することで C と同様に使えるのはよいとしても、`gzopen()` や `gzclose()` などは C でコンパイルしたライブラリ（`zlib`）の中にある関数です。これらが C でコンパイルされていることをコンパイラに伝える必要があります。C++ から C++ 以外の言語でコンパイルされた関数を使うには linkage specification を使って宣言します。例えば `gzopen()` が C でコンパイルされていることを宣言するには、

```
extern "C" gzFile gzopen(const char*, const char*);
```

とします。また、複数の宣言をまとめて行いたい場合、例えば `gzclose()` も含めたい場合、次のように`{}` でブロック化することができます。

```
extern "C"
{
        gzFile gzopen(const char*, const char*);
        int gzclose(gzFile);
}
```

しかし プログラム 14.15 には、どこにもそのような宣言はありません。この宣言は、`zlib` の場合、`zlib.h` に書かれてます。このヘッダファイルを見ると、最初の方に

プログラム 14.15: 簡単な gzip ファイル作成例

```
1  #include <iostream>
2  #include <cstdio>
3  #include <zlib.h>
4
5  static const int IOBUFSIZE = 1024;
6  char buf[IOBUFSIZE];
7
8  int
9  main(int argc, char* argv[])
10 {
11         if (argc != 3)
12         {
13                 std::cerr
14                         << "Usage: " << argv[0]
15                         << " sourcefile destfile" << std::endl;
16                 return (-1);
17         }
18
19         FILE* fsrc = fopen(argv[1], "rb");
20         if (fsrc == NULL)
21         {
22                 std::cerr
23                  << "ERROR in fopen(" << argv[1] << ")" << std::endl;
24                 return (-1);
25         }
26         gzFile fdst = gzopen(argv[2], "wb");
27         if (fdst == NULL)
28         {
29                 std::cerr
30                  << "ERROR: in gzopen(" << argv[2] << ")" << std::endl;
31                 fclose(fsrc);
32                 return (-1);
33         }
34         int n;
35         while ((n = (int)fread(buf, 1, IOBUFSIZE, fsrc)) > 0)
36                 if ((int)gzwrite(fdst, buf, n) != n)
37                 {
38                         std::cerr << "ERROR in gzwrite()" << std::endl;
39                         break;
40                 }
41         gzflush(fdst, Z_FINISH);
42         fclose(fsrc);
43         gzclose(fdst);
44         return 0;
45 }
```

```
#ifdef __cplusplus
extern "C" {
#endif
```

とあり、一番最後付近に

```
#ifdef __cplusplus
}
#endif
```

と書かれています。ここで、__cplusplus は C++ の規格で定められているシステム定数で、標準規格準拠の C++ 環境では必ず定義されています。つまり、このヘッダを C++ で使う時には全体が extern "C" で linkage specification され、C で使う時には、この linkage specification ははずされます。従って、C で使う時も C++ で使う時もユーザはこのヘッダファイルを使うだけで済むようになっています。

C と C++ 両方での利用を意識したライブラリなどでは、ヘッダはこのように書かれているのが普通です。

もし、古い C ライブラリなどで、ヘッダがこのように書かれていない場合は、ヘッダを書き換える必要がありますが、ライブラリ管理の都合などでヘッダ自身を書き換えるのが困難な場合は、利用する側のソースで linkage specification を行います。例えば oldlib.h がそれだとすると、

```
extern "C" {
#include "oldlib.h"
}
```

のように #include の前後を extern "C" でくくります。

14.4.2 gzip ファイルストリームを作る

前節で見たように、C++ では C の記法が使えるので、ほぼ C の使い方のまま C++ でも使うことができます。一方 C のライブラリはそのままにして、C++ の様々な便利は機能を使いたいこともしばしばあります。例えば、前節の gzip ファイルの場合、C++ のオブジェクトであれば、わざわざ gzclose() しなくてもスコープから抜けた時点で自動的に閉じることもできます。そこで、ここでは gzip ファイルを C++ の入出力ストリームとして使えるようにしてみましょう。

C++ の標準テンプレートライブラリの一つである入出力ストリームやそれに関連するライブラリは、このような要求がたやすく実現できるように設計されています。まずは、この設計がどのようになされているのかを説明します。

バッファ付き入出力の担当クラス

C++ の標準ライブラリとして用意されている std::iostream クラスは実は直接的には入出力装置とデータをやりとりしていません。入出力装置とのデータのやりとりを担当しているのは std::streambuf （とその派生）クラスです。このクラスはバッファ付きの入出力を行うように設計されています。

一方、std::iostream クラスのコンストラクタは、この std::streambuf へのポインタを引数として要求します。つまり、std::iostream は std::streambuf へのポインタを通じて、その入

出力関数を使ってデータの入出力を行います。逆に言えば std::streambuf の派生クラスを作ってその入出力関連の関数を用意すれば後はわずかな実装で入出力ストリームクラスができます。

更に、バッファ付き入出力担当の基底クラスである std::streambuf は派生クラスでの実装が容易に行えるよう設計されています。実際、派生クラスでやるべきことは、

- コンストラクタで装置を使えるようにし、バッファを用意してその開始位置と終端位置のポインタを基底クラスに通知する。
- 読み込みバッファが空になった時に呼ばれる関数を実装する。すなわち、装置からバッファにデータを読み込み、データが詰められたバッファの開始位置と終端位置を基底クラスに通知する。
- 書き込みバッファが一杯になった時に呼ばれる関数を実装する。すなわち、装置へバッファの内容を書き出し、書き込み可能になったバッファの開始位置と終端位置を基底クラスに通知する。
- デストラクタで書き込みバッファにデータが残っていれば装置に書き出し、装置を閉じる。

細かい制御を行うには、他にもいくつかの関数を実装する必要がありますが、基本的にはここに示した実装だけで使えるようになります。

バッファ付き入出力クラスの設計

基底クラスである std::streambuf から派生させ、どのような機能を実装すればよいかを述べたので、次はもう少し詳細に検討してみます。特に装置の使用準備を行う、すなわち今の場合はファイルを開く機能をどこにどのように持たせるかが、このクラスの利用の仕方を決める重要な点です。

幸いにも、今の場合はお手本になるクラスとして標準ファイルストリームクラス、std::fstream がありますので、これを見てみましょう。

このクラスはファイル入出力を扱いますが、実際の入出力を扱っているのは std::streambuf の派生クラスであるファイルバッファクラス、std::filebuf です。規格を調べると、このクラスの公開メンバ関数には std::streambuf クラスから継承したメンバ関数の他に（コンストラクタとデストラクタを除いて）3 つの関数があります。またコンストラクタは引数無しです。このことから、このクラスではファイルの使用にあたっては

1. まず、ファイルバッファオブジェクトを構築し、
2. ファイルを open することによりファイルの実体と結びつける。

という二段階の手順を踏むということがわかります。また、同じファイルバッファオブジェクトを使って、close() した後、再度 open() を呼び出すことにより別のファイル実体と結びつけることも可能です。つまり、このクラスの設計思想は、このクラスは、あくまでもバッファであって、ファイル実体と結びつけられているか否かは、そのバッファの状態の一つに過ぎないという考え方です。

もちろん、別の設計思想でバッファクラスを作ることもできます。例えばこのクラスをファイル実体そのものを表現したものであるとする考え方もあるでしょう。この場合、コンストラクタでファイル実体を開き、デストラクタで閉じるという実装も可能です。ファイル実体の無い、あるい

は実体を作ることのできないものは、最初からオブジェクトとして構築できないとする考え方です。オブジェクト指向という意味からは、むしろこちらの方が素直かもしれません。

しかし、C++ は設計思想を強制したりはしません。様々な設計思想を表現できるプログラミング言語です。気をつけないといけないのは、設計思想が決まると、それに応じて持つべき機能やデータが決まってくるということです。つまり一度設計思想を決めてプログラミングを始めて、進めば進むほど後戻りは大変になるということです。設計段階での見落としに気づかずに、それが後戻りを余儀なくさせることもあります。そこで、ここでは単純に標準ライブラリのファイルバッファクラスを真似ることにしましょう。

ファイルバッファクラス

さて、それではファイルバッファクラスを見てみましょう。前節で述べたようにこのクラスはファイル実体とは独立しています。バッファとファイル実体を結びつける関数が `open()` であり、このバッファに結びついているファイル実体を切り離すのが `close()` です。

規格を見ると、もう一つ重要な仕様があります。それは、このバッファとファイル実体が結びついているか否かを `bool` 値で示す `is_open()` という関数があり、`is_open()` が `true` を返す時に `open()` を呼び出すと失敗するという仕様です。このことから、このバッファに結びつけることができるファイル実体は高々一つであることがわかります。ただし `open()` と `close()` の対を繰り返す使うことはできますので、同じバッファで結び付けられているファイルを、異なるファイル実体に次々と切り替えていくことはできます。

```
g++ gztest2.cpp gzfilebuf.cpp -lz
```

プログラム 14.16: GZfilebuf ヘッダ

```
1  // gzfilebuf.h
2  #ifndef GZFILEBUF_H_INCLUDED
3  #define GZFILEBUF_H_INCLUDED
4
5  #include <streambuf>
6  #include <zlib.h>
7
8  namespace mylibrary
9  {
10
11 class GZfilebuf : public std::streambuf
12 {
13 public:
14         GZfilebuf(const char* filename, std::ios_base::openmode mode);
15         virtual ~GZfilebuf();
16
17 protected:
18         virtual int overflow(int c = std::char_traits<char>::eof());
19
20 protected:
21         static const int GZFILEBUFSIZE = 8192;
22         std::ios_base::openmode m_openmode;
23         gzFile m_gzfile;
24         char m_buf[(GZFILEBUFSIZE + 1)];
25 };
26
27 }       // namespace mylibrary
28
29 #endif  // GZFILEBUF_H_INCLUDED
```

プログラム 14.17: GZfilebuf 関数定義

```
1  // gzfilebuf.cpp
2
3  #include "gzfilebuf.h"
4
5  namespace mylibrary
6  {
7
8  GZfilebuf::GZfilebuf
9          (const char* filename, std::ios_base::openmode mode) :
10                 m_openmode(mode), m_gzfile(NULL)
11 {
12         m_gzfile = gzopen(filename, "wb");
13         setp(m_buf, m_buf + GZFILEBUFSIZE);
14 }
15
16 GZfilebuf::~GZfilebuf()
17 {
18         if (m_gzfile)
19         {
20                 overflow(std::char_traits<char>::eof());
21                 gzflush(m_gzfile, Z_FINISH);
22                 gzclose(m_gzfile);
23         }
24 }
25
26 int
27 GZfilebuf::overflow(int c)
28 {
29         if (m_gzfile == NULL)
30                 return std::char_traits<char>::eof();
31
32         std::streamsize n = pptr() - pbase();
33         if (n > 0)
34         {
35                 if ((std::streamsize)gzwrite(m_gzfile, pbase(), n) != n)
36                         return std::char_traits<char>::eof();
37         }
38         setp(m_buf, m_buf + GZFILEBUFSIZE);
39         if (c == std::char_traits<char>::eof())
40                 return 0;
41         *m_buf = c;
42         pbump(1);
43         return c;
44 }
45
46 }       // namespace mylibrary
```

プログラム 14.18: GZfilebuf を使った gzip ファイル作成例

```
1  #include <iostream>
2  #include <fstream>
3  #include "gzfilebuf.h"
4
5  static const int IOBUFSIZE = 1024;
6  char buf[IOBUFSIZE];
7
8  int main(int argc, char* argv[])
9  {
10         if (argc != 3)
11         {
12                 std::cerr
13                         << "Usage: " << argv[0]
14                         << " sourcefile destfile" << std::endl;
15                 return (-1);
16         }
17
18         std::ifstream ifs(argv[1], std::ios::binary);
19         if (ifs.fail())
20         {
21                 std::cerr
22                  << "ERROR in ifs(" << argv[1] << ")" << std::endl;
23                 return (-1);
24         }
25         mylibrary::GZfilebuf ofb(argv[2], std::ios::in);
26         std::ostream ofs(&ofb);
27         if (ofs.fail())
28         {
29                 std::cerr
30                  << "ERROR: in ofs(" << argv[2] << ")" << std::endl;
31                 return (-1);
32         }
33         std::streamsize n;
34         do
35         {
36                 ifs.read(buf, IOBUFSIZE);
37                 if((n = ifs.gcount()) > 0)
38                 {
39                         if(!ofs.write(buf, n))
40                         {
41                                 std::cerr << "ERROR in gzwrite()"
42                                         << std::endl;
43                                 break;
44                         }
45                 }
46         }
47         while (ifs);
48         return 0;
49 }
```

第15章　おわりに代えて

本書の冒頭に述べたように、本書は技術職員専門課程研修の講義録です。本講義の記録と感想をここに記して、おわりに代えたいと思います。

15.1　研修内容について

本研修は、講義と実習の形をとりました。これは平成 8 年に行なった同様の研修では講義だけだったために聴講者から難しい、わかりにくいなどの意見があったためです。実習を行なうとなると、そのため、Windows、Linux、MacOS 共通に使えるプログラム開発環境として g++ を選びました。聴講者は 11 名（受講者 7 名、聴講のみ 4 名）で全員ノート PC 持参での実習でしたが、実習そのものに特にトラブルはなかったように思います。

研修は 1 回 2 時間で全 10 回、これで C++入門を行なうというのはかなり無謀であることは十分に予想されました。しかし実際問題として、プロのプログラマでも規格を全部理解しているとはとうてい思えないし、ましてやプロでもない私が教えるわけなので、むしろ実際のプログラムを書かなければいけない状況にどう向き合うかという視点から必要最小限の部分を選び出し、どういう考え方でプログラムを書けばよいか、人のプログラムを読む時に、どういう理解の仕方をすればよいかという点を強調することに重点を置いたつもりです。

公募案内の内容は以下のようなものでした。

研修講座 C++入門

研修期間 全 10 回 2010 年 6 月 3 日（木）ー 8 月 5 日（木） 週 1 回 2 時間

会場 2 号館 1 階会議室（中）など

講師 藤井啓文（素粒子原子核研究所 教授）

受講者 技術系職員を対象とし、技術調整役の推薦に基づき、機構長が認めた者。および業務で C++のプログラミングに携わる方や興味をお持ちの方で、技術調整役の推薦に基づき、機構長が認めた者。

概要 C++ はプログラミング言語 C との互換性を強く意識して作られたコンパイル型（プログラムを書いた後、言語処理プログラムで機械語に翻訳させ、実行用のファイルを作る）の言語です。実行の高速性を求められる場合や、メモリーや周辺デバイスなどのコンピュータ資源を細部にわたり制御しながら利用したい場合などによく用いられます。その分、敷居も高く、また危険な落とし穴も数多く存在し、利用には知識と経験が要求されます。C++ は膨大な言語です。本講義は、その入門として、データ処理を念頭に置いて、そこで必要となる知識や落とし穴を中心に解説していきます。

世話人 仲吉一男（素粒子原子核研究所 技師）

実際に行なった内容は以下のようなものでした。

- 事前準備：各自のノート PC で C++ の開発環境を用意しておくこと。Windows ユーザで、用意できていない人は、研修用 Web ページを見て、準備すること。
- 第 1 回（6 月 3 日 9:30-11:30 2 号館 1 階中会議室）まずは動かしてみる。
- 第 2 回（6 月 9 日 13:00-15:00 研究本館 1 階会議室 2（北））入出力と整形。
- 第 3 回（6 月 17 日 9:30-11:30 研究本館 1 階会議室 2（北））算術型と型変換。
- 第 4 回（6 月 24 日 9:30-11:30 2 号館 1 階中会議室）配列とポインタ。
- 第 5 回（7 月 1 日 9:30-11:30 2 号館 1 階中会議室）単純なクラス。
- 第 6 回（7 月 8 日 9:30-11:30 2 号館 1 階中会議室）分割コンパイル。名前空間、演算子のオーバーロード、派生と継承。
- 第 7 回（7 月 15 日 9:30-11:30 2 号館 1 階中会議室）テンプレート、入出力ストリームの利用。
- 第 8 回（7 月 22 日 9:30-11:30 2 号館 1 階中会議室）動的記憶域確保・解放、デストラクタ、コピーコンストラクタ、代入演算子。
- 第 9 回（7 月 29 日 9:30-11:30 2 号館 1 階中会議室）事象処理の実装例
- 第 10 回（8 月 5 日 9:30-11:30 2 号館 1 階中会議室）実装例の続きと例外処理。

15.2　終了後のアンケート結果

終了後に主催者により行なわれたアンケート結果を掲載しておきます。アンケート回答にもありますが、本研修の世話係として快く会場の手配や準備をして下さった仲吉一男氏に感謝の意を表して本書をしめくくりたいと思います。

「C++入門」受講後のアンケート

「C++入門」受講者の皆様

「C++入門」の受講お疲れさまでした。今後の専門研修の参考のためアンケートにお答えくださるようお願い致します。講師その他、研修委員会へは結果をまとめた形で（個人を特定しない）提出しますので、忌憚のないご意見をお願い致します。

■受講前の知識について

1. C 言語の知識について
    a. プログラミングできる（仕事で使っている）　40%
    b. プログラミングできないがソースコードは理解できる　40%
    c. 使ったことがない　20%

2. C++の知識について

a. プログラミングできる（仕事で使っている） 10%
b. プログラミングできないがソースコードは理解できる 30%
c. 使ったことがない 60%

■ C++入門受講後

3. 全体の印象について
a. 予想通りの講義と実習だった 40%
b. だいたい予想通りだった 30%
c. 予想と違っていた 30%
d. まったく違っていた 0%

4. 自己評価
a. 簡単なプログラミングはできるようになった 20%
b. 参考書を見ながらならできる 30%
c. ソースコードはある程度理解できるようになった 40%
d. 受講前と変わらない 10%

■講義内容と進行

5. 講義内容について
a. やさしかった 0%
b. ふつう 50%
c. 難しかった 50%
d. 非常に難しかった 0%

6. 講義の進行について
a. 遅かった 0%
b. ふつう 40%
c. 早かった 40%
d. 非常に早かった 10%
e. 後半速くなった 10%

コメント：

- 実習中コンパイルエラーを起こすとそこで終わり。
- 前半はちょうど良いか遅いぐらいであったが、後半はかなり速く感じた。

■実習

7. 実習時間は十分でしたか
a. 長すぎる 0%
b. ちょうどよかった 30%
c. 足りなかった 70%

コメント：

- 私はキー入力が遅いため、ついていけなかった。キー入力することが目的になりつつあったので、キー入力はあきあらめ、コードを理解することと、講師の話を聞くことに注力した。
- 講義と実習の時間が明確に分かれていない。講義の進むスピードは問題ないと思うが、受講者がコードを書く時間があまりなかった。たとえば 15 分講義を行い 5 分コードを書く時間を与える。これの繰り返しで進めていけば良いと思う。もちろん内容によってはこの限りではないが。
- 実用になる演習プログラムは良かったと思います。音楽ファイルが自分のプログラムから作れることがわかり、応用範囲が広がりました。
- 甘えですが、同じ内容で再開講していただけると理解が深まります。
- 実習前のスケジュールを見ると長く感じたが実際受けてみると足りなかった。

■その他

8. 今後仕事で C++ を使う可能性がありますか?
    a. ある 50%
    b. ない 0%
    c. 使っている 40%
    d. わからない 10%

9. C++関連の続編をやるとすると、どのような内容を期待しますか?
    a. 今回と同じ内容 20%

コメント：

- もっと詳細に
- 今回最後は音声ファイルの扱いでしたが、計測制御インタフェースの扱いなど

    b. 今回やらなかった内容 40%
    c. より高度な内容 60%
    d. 必要ない 0%

コメント：

- また楽しい演習プログラムを期待します。

10. 今後公開を前提に毎回ビデオ撮影を行いました。それに関連して
    a. 有効だと思う 50%

理由：

- 出席できなかった回の復習ができる。講義のレベルが自分の目で確認できる。
- 講義に欠席した時の補習用として。
- 良い内容であったので、後で見返せるし、受けなかった人も見れる。
- 復習できるので。

- できればサーバー上のコンテンツとしてブラウザー等から見れれば嬉しいです。
- スクリーンを写しているだけだったので、PC の画面自体を取り込めるソフトを利用し、音声をつけたほうが有効に思えた。

b. 必要ない 20%

理由：

- 実習することに意味があると思います。講義を聴くだけなら参考書だけで良いのではないでしょうか。
- 実際に講義に出席しながら学習するのに意味があるのであって、ビデオでの学習には大きな効果は期待できないからである。特に C++は関連書籍も充実しているので、ビデオで学習するよりこれらのテキストを利用した方が効果が高いと思われる。

c. わからない 30%

11. C++入門の講義ビデオがありますが、公開された場合観たいですか?
    a. ぜひ観たい 30%
    b. 観たい 40%
    c. 必要ない 30%

コメント：

- 実習することに意味があると思います。講義を聴くだけなら参考書だけで良いのではないでしょうか。

12. 講師に一言どうぞ

- ありがとうございました。いくつかのソースコードと実行画面が同時に見ることが出来ると見比べながら講義をきけるとよかったなと思います。(早すぎるとき、画面切り替えでどこのことをいってるのかわからなくなる時があったので)。
- 長い間おつかれさまでした。大変参考になる内容でした。続編があるようでしたら聴講したいと思います。
- 暑い中ありがとうございました。
- 長期間ありがとうございました。
- オブジェクト指向のプログラミングについて、入口ではありますが、理解できました。ありがとうございました。
- ありがとうございました。
- ご苦労様でした。今後もまたお願いします。
- 講義をして頂きありがとうございました。また各種調整業務をして頂いた仲吉さんにも感謝したいと思います。
- 教材準備、毎週の講義など大変だったと思います。ありがとうございました。

- 概念的な部分からプログラミングのこつまで聞けて非常にためになる講義でした。過去にC言語に挑戦してポインタでつまづいて諦めた事があり、今回の講義でなんとか概念が理解できました。ありがとうございました。

13. その他なんでもどうぞ

- 実践にすぐに対応できそうな「入門編」だったのが良かったです。
- C++をやるきっかけになりました。どうもありがとうございました。
- 藤井先生が現在執筆中の「データ処理のためのC++入門」の製本版が完成したら欲しいです。
- 仲吉さんもお疲れ様でした。
- お世話係の仲吉様、ご苦労様でした。

付 録A　規格について

A.1　国際規格と JIS

C++ は

ISO/IEC 14882:2003, Information Technology--Programming languages--C++

という国際規格があります。日本では、これを基礎にして

JIS X 3014:2003 プログラム言語C＋＋

という日本工業規格を定めています。

日本工業規格については、読むだけであれば日本工業標準調査会の Web ページ

http://www.jisc.go.jp/

にあるデータベース検索を使って調べることができます。

A.2　プログラム言語 C との関係

プログラム言語 C++ はプログラム言語 C に対し上位互換になるよう開発されました。一方 C 自身も C++ をはじめとする他のプログラム言語からの影響も受けながら規格を拡張してきています。そこで、C の規格についても少し述べておきます。

普通に C 言語と呼ばれているのは多くの場合、いわゆる ANSI-C とか C89 または C90 と呼ばれる

- ANSI X3.159-1989 Programming Language-C
- ISO/IEC 9899:1990 Programming languages C
- JIS X 3010-1993 プログラム言語C

の規格を指します。

これにワイド文字などの追補を含んだ C95 と呼ばれる

- ISO/IEC 9899:1995/Amd1:1995 (Programming languages - C, AMENDMENT 1: C integrity)
- JIS X3010:1993/AMENDMENT 1:1996　プログラミング言語C（追補 1）

の規格があります。

更に、C99 と呼ばれる

- ISO/IEC 9899:1999 Programming languages C
- JIS X3010:2003　プログラム言語C

の規格があります。

このように、C と C++ は独立して規格化されています。従って、上位互換とは言っても、完全に上位互換というわけではありません。とは言え、C++ は C の上位互換であることを強く意識して設計された言語ですので、大抵の C のプログラムは修正無しか、ごく僅かな修正で動かすことができます。

A.3 ソースファイル

A.3.1 基本ソース文字集合

C++ では、どのような処理系であってもソースファイル中で使える文字集合を基本ソース文字集合（basic source character set）として定め、水平タブ（HT）、垂直タブ（VT）、改ページ（FF）、改行（LF）、空白（SP）と以下の図形文字

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = ,
\ " '
```

の合計 96 文字としています。ただし、C++ では翻訳過程の第一段階で、必要に応じてソースファイル中の文字を基本ソース文字集合の文字へ変換することができるようになっており、その変換は処理系依存です。例えば Windows のテキストファイルでの行末は通常、復帰（CR）と改行（LF）の対ですが翻訳過程の第一段階でこの復帰と改行の対が改行一つに置き換えられます。また、処理系によっては日本語を含めることができるものもあります。ですが、処理系によらないプログラムを書く場合には、基本ソース文字集合の文字のみを使うのがよいでしょう。

A.3.2 ソースファイルの終端

ソースファイルの終端は空行である必要があります。すなわち改行のみで終わっていなければなりません。改行のみで終わっていない場合は、通常は警告が出ます。

A.3.3 空白文字

空白文字は語を区切るのに使われます。語の区切りが明確に判断できる場所であれば無くても構いません。また語を区切る場所であれば、いくつあっても構いません。改行や水平タブも空白文字とみなされます。行の先頭は、その前に改行があるわけですから、行の先頭に空白を入れなくとも語の先頭になります。つまり、プログラム 2.2 の

```
        std::cout << "hello, world" << std::endl;
        return 0;
```

という部分は

```
std::cout << "hello, world" << std::endl; return 0;
```

と書いても、

```
                std::cout <<
                   "hello, world"
<< std::endl;
      return 0;
```

と書いても、文法的には全く同じです。

では

```
int main()
```

の部分を考えてみましょう。関数の名前には () は使えません。ですから main() と書いても関数の名前が main であることは明確に判断できるので、main と () の間に空白は必要ありません。もちろん空白文字を入れて main () としても構いません。一方 int main の方は、空白を消すと intmain となり、intmain という名前になってしまいますので、int と main の間の空白は必要です。ただし改行は空白文字と同じ扱いですので、

```
int
main()
```

とすることは可能です。

ここで述べた改行や空白文字の規則からプログラム 2.2 は、びっしり詰めて

```
#include <iostream>
int main(){std::cout<<"hello, world"<<std::endl;return 0;}
```

と書くこともできます。

A.4 識別子とスコープ

A.4.1 識別子

C++ では、変数名や関数名などを名前で区別します。プログラムやライブラリの書き手が与えるこの名前を識別子（identifier）と言います。識別子として使える文字は（半角英数文字の）アルファベット、数字およびアンダースコア（_）です（規格では他にも使える文字がありますが、使わない方が無難です)。識別子の先頭文字には数字は使えません。また、アンダースコアで始まる識別子はシステムで予約されています。

識別子の表す型など記憶域上の実体を伴わない情報を言語処理系に伝えるものを宣言と呼びます。言語処理系は、この情報を元にソースを翻訳していきますので、一つの翻訳単位は、その中で使われるすべての識別子の宣言が含まれていなければなりません。ライブラリで使う識別子などは多くの場合インクルードファイルを通じて宣言されています。

識別子に対し、記憶域上での実体に関する情報や関数の本文などの情報を言語処理系に伝えるものを定義と呼びます。一つのプログラムで使用する関数やオブジェクトの定義は一つでなければなりません。

A.4.2 スコープ

一つ識別子の宣言が与えられた時に、その宣言が有効な領域をその識別子のスコープと呼びます。スコープは入れ子にすることができて、内側のスコープで宣言された識別子は外側のスコープからは見えません（隠蔽されます)。

A.4.3 名前空間

複数のライブラリなどを使っていると識別子が競合することがあります。名前空間はスコープに名前を付けて管理することで、この問題を解決します。

表 A.1: 予約語

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
go	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

表 A.2: 代替語

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

A.5 予約語

表 A.1 に示す語は予約語（キーワード、keyword）として、特別な意味が与えられています。別の意味の識別子として用いることはできません。(規格書 2.11)。

また、表 A.2 に示す語は、演算子の記号などを文字で代替表現する場合に使います。これらも別の意味の識別子として使うことはできません。

A.6 演算子と区切り子

表 A.3 に示す記号や記号の組み合わせ、及び語は演算子 (operator) または区切り子 (punctuator) として使われます。これらは翻訳にあたり語を区切る役割も果たします（規格書 2.12)。語を区切っていく時に複数の候補がある場合は、一致する文字や記号の個数が最大のものが選ばれます。例えば

```
int a(3), b(4);
int c = a+++b;
```

は、

```
int a(3), b(4);
int c = a++ +b;
```

として翻訳されます。つまり、実行後は a と b の値は 4 に、c は 7 になります。当然ながら、

```
int a(3), b(4);
int c = a+ ++b;
```

表 A.3: 演算子と区切り子

{	}	[	]	#	##	(	)	
<:	:>	<%	%>	%:	%:%:	;	:	...
new	delete	?	::	.	.*			
+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	,	->*	->
and	and_eq	bitand	bitor	compl	not	not_eq		
or	or_eq	xor	xor_eq					

として空白文字で区切りを明示すれば、実行後は a は 3、b は 5、c は 8 になります。

A.7　式と演算子

A.7.1　左辺値と右辺値

式や演算子による操作を述べる時に重要な概念として左辺値（lvalue）と右辺値（rvalue）があります。左辺値というのは、記憶域の位置により参照される値です。記憶域が割り当てられているので、アドレス演算子 & のオペランドとすることができます。また定数を意味する const 修飾子が付与されていなければ、代入演算子 = の左オペランドとすることができます（元々の意味はここからきています)。

一方右辺値というのは値そのものです。必ずしも記憶域が割り当てられるとは限りません。

例をあげましょう。

```
int a, b;
  :
a = b + 1;
```

この場合の、一番最後の代入文は a という変数に割り当てられた記憶域の内容を右辺の値にせよということですので、a は左辺値です。一方、右辺の式の値は例えば記憶域とは別の演算レジスタの値がそのまま a で示される記憶域へ代入されるかも知れません。整数型のオブジェクトが一時的に作られそこに結果の値が格納された後、a と名前付けられた記憶域にその値が代入されるかも知れません。いずれにせよ結果の値だけが問題で、それが記憶域にあるか無いか、あるいはどこの記憶域であるかは問題ではありません。つまり右辺値です。

付 録B　基本データ型とその表現

C でも C++ でもデータの内部表現までは規定していません。しかしながら、装置からデータを読み取る場合とか、異なる OS 間でデータを交換する場合など、内部表現を知らなければならない場合があります。このような問題に対し、実際にはある程度標準的に用いられている表現が存在します。基本データ型について以下に代表的な表現を述べます。

B.1　整数の内部表現

整数（char 型も含む）の内部表現は通常 2 進数表現が使われます。問題となるのは負の数の表現ですが、現在では 2 の補数表現を使うのが一般的です。

一般に N 進数 n 桁の表現があった時、N^n にするために補うべき数（すなわち N^n からの差）を N 進数の N の補数表現、また $N^n - 1$ からの差を N 進数の $(N-1)$ の補数表現と呼びます。従って 2 進数表現の場合、2 の補数表現と、1 の補数表現があります。$(N-1)$ の補数表現を得るには桁ごとに $(N-1)$ からの減算で済みます。2 進数における 1 の補数表現はビット毎の反転で済み、2 の補数表現はそれに 1 を加えたもの（最上位ビットが桁あふれしても無視する）です。N の補数表現を用いることで、正整数間の減算処理を加算処理に置き換えることができます。

B.2　浮動小数点数の内部表現

現在の C/C++ の多くは浮動小数点数の内部表現として IEEE-754 (IEEE Std 754-2008 IEEE Standard for Binary Floating-Point Arithmetic) という規格の基本形式（basic formats）を用いており、`float` はその `binary32` 形式を、`double` は `binary64` 形式を用いています。これらはいずれも IEEE-754 の交換用形式（interchange formats）の一つでもあります。

これらの形式でのビットの並びは、いずれも最上位の 1 ビットは符号ビット (S) で以下指数部 (Exp)、仮数部 $(Fraction)$ と並びます。最上位ビットを左端に配列した場合の並びを図 B.1 に示します。

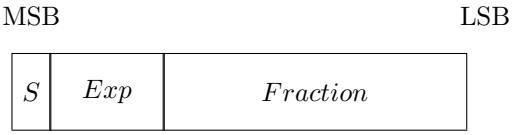

図 B.1: IEEE-754 2 進数形式でのビットの並び

指数部の全ビットが 1 でかつ仮数部の全ビットが 0 の場合は無限大を表します。符号ビットも有効で、正の無限大と負の無限大があります。

指数部の全ビットが 0 でかつ仮数部の全ビットが 0 の場合は 0 を表します。ここでも符号ビットが有効で正の 0 と負の 0 が存在します。浮動小数点数の 2 進数表現は近似値なので、負数から丸められて 0 になったのか、正数から丸められて 0 になったのか区別する必要がある場合などに有効です。ただし通常の数値比較ではこの二つの 0 は区別されません。

指数部の全ビットが 1 でかつ仮数部に 0 ではないビットが 1 つでも存在する場合、非数 (`NaN`) を表します。これは演算結果が数として表現できない場合、例えば実数空間内で負数の平方根をとった場合などを表すのに使われます。

上記以外で、指数部が 0 でない場合、正規化数と呼び、指数部 (Exp) を符号無し整数として

$$(-1)^S \times 1.Fraction \times 2^{(Exp-bias)}$$

の数値を表します。ここで $bias$ は `binary32` 表現の場合は 127、`binary64` 表現の場合は 1023 である。$Fraction$ の前に 1 が付加されていることに注意います（いわゆるケチ表現)。

指数部が 0 で、仮数部が 0 でない場合、非正規化数と呼び、

$$(-1)^S \times 0.Fraction \times 2^{(1-bias)}$$

の数値を表します。$Fraction$ 部の前が 0 になり、それに伴い正規化数の場合と比べ指数部が 1 だけ増えていることに注意します。

B.3 文字の内部表現

文字を表す `char` 型は、これまでにも述べたように整数型の一部とみなすことができます。各文字には整数番号が割り当てられる。この時、機種やシステム毎に割り当ててある番号が異なると、通信やファイルの交換などに非常に手間がかかります。これを軽減するために、文字に割り当てる番号（文字コード）が C/C++ とは別に規格化されています。残念ながら現時点では規格は一つではありません。このため、いわゆる「文字化け」が起きたりします。

ただし、英数字といくつかの（英語圏で用いられる）記号については、American Standard Code for Information Interchange (ASCII) 規格を用いるのが一般的です。

文字について、もう一つ注意すべきは、改行や復帰、タブなど見えない文字にもコードが割り振られていることです（これらのコードを制御コードと言います)。

B.4 拡張文字

ここでは、日本語など、いわゆる半角英数文字以外の文字を扱う際に必要になる事項について述べます。

B.4.1 ワイド文字（`wchar_t` 型）

規格上は `char` 型の格納サイズは 8 bit 以上とされています。8 bit あると原理的には 256 個の文字を表現することができます。日本語などを含めようとするともちろん 8 bit では足りません。

では、規格上許されているからというので char 型を 8 bit より大きくした（例えば 16 bit にした） C/C++ コンパイラはあるかというと少なくとも筆者の知る限りありません。

実際問題として、現時点（2010 年初頭）では多くの計算機やその周辺装置、通信などでは処理の最小単位は 8 bit になるよう設計されています。多くのプログラムで char 型やその配列型は、8 bit 単位でのデータをやりとりする型として使われています。つまり今更これを簡単に 16 bit などに変更できない状況にあるわけです。そこで char 型は現状のままで、16 bit 以上の格納サイズを持つ文字型として wchar_t 型が導入されています。

B.4.2 文字コード集合

文字の数値表現は統一されてないと情報交換に困ります。ある計算機ではアルファベットの A は 65 であるが、別の計算機では 193 であるというのでは通信も困るし、ファイル交換でも困ります。そこで、よく使われる文字のコードは規格として割り振ってあります。米国でよく使われる文字は American Standard Code for Information Interchange (ASCII) として規格化されています。この規格では 0 から 127 までの数値に制御文字を含む文字が割り振られ、33 から 126 までが可読文字に割り当てられています。これに対応する形で ISO/IEC 646 という国際規格が定められ、これに準拠した形で各国の規格があります。例えば日本では JIS X 0201 がこれに該当します。ISO/IEC 646 では各国で決めてよい部分というのがあり、例えば 92 は、ASCII では backslash が、JIS では円記号が割り振られています。

格納サイズが 8 bit であっても、上記規格ではまだ 128 から 255 まで空いています。欧州では、ここに Latin-1 と呼ばれる欧州各国で用いられる文字を割り当てています。日本では上記 JIS X 0201 として、カタカナ（いわゆる半角カタカナ）と若干の記号を割り当てています。

B.4.3 統一文字コード

インターネットが発達し、情報交換が盛んになると文字コード集合を国に合わせて切り替えるのでは手間がかかります。また、日中露 3ヶ国語辞典の編集などをやろうすると非常に面倒になります。そこで、当然の流れとして世界中の文字に統一コードを割り当てようとする動きが起こりました。こうして生まれたのが Unicode と ISO/IEC 10646 です。

Unicode は当初、格納サイズ 16 bit 固定長にすべての文字を収めるという、漢字文化圏の人間から見ると極めて無謀な計画でスタートし、案の定 Unicode version 1 は 5 年で破綻し Unicode version 2 で大幅な改正が加えられました。現在の Unicode の格納サイズは 21 bit で、16 bit で表現する時は一部を surrogate pair と呼ばれる 16 bit 2 単位を使って表現します。つまり固定長ではありません。また、version 1 と version 2 では、いわゆる「ハングル文字の大移動」があり、同じコードであっても異なる文字が割り当てられている部分があります。

一方 ISO/IEC 10646 は、当初、既存の文字コード集合との互換性を最大限持たせたまま拡張するという方針を採り Universal Character Set (UCS) と呼ばれる集合を提案しました。既存コード集合との互換性を最大限持たせるため、基本的には group、plane、row、column の 4 つのオクテットで表現する、すなわち 32 bit を必要とすることとなりました（group は 7 bit なので実際には 31 bit）。この案は Unicode グループからの反対もあり、採択されず、結局 Unicode の 16 bit を group 0、plane 0 の Basic Multilingual Plane (BMP) として割り当て、group plane を省略した 2 オクテット表現を認め、UCS-2 として定める（4 つのオクテットで表現する方は UCS-4 と呼ぶ）ことで採択されました。

その後、Unicode が 16 bit 固定長で破綻をきたすと、ISO/IEC 10646 の plane の概念を利用できるよう surrogate pair が導入され、16 bit 固定長で収納できなかった文字は追加面に収納できるようになり、これ以降 ISO/IEC 10646 と Unicode は歩調を合わせています。従って、現在では Unicode と ISO/IEC 10646 は、ほぼ同一であると見なしてよくなっています。

B.4.4 文字 encoding

文字コード集合が決まっても、文字に対するコードがそのまま計算機上でその値の数値になっているとは限りません。規格に定められたコード値そのままでは、計算機にとって不都合なこともあります。そのような場合、機械的に処理できる（1 対 1 対応で、順番も変えない）方法でコードを変換して用いることがあり、これを文字 encoding と呼びます。

例をあげます。JIS X 0208 はアルファベット、ひらがな、かたかな、漢字や数学記号なども含む 2 オクテットの文字コード集合です。実際には、JIS X 0201 と共存させるために、これがそのまま用いられることはなく、

- Shift_JIS
- EUC-JP

などの encoding が用いられています。ついでに言うと、JIS X 0208 は、それ単独で完全な文字コード集合であり、JIS X 0201 に含まれるアルファベットや数字、カタカナを含んでいます。上記の encoding は、このことを無視して JIS X 0201 を共存させたために、JIS X 0201 にある文字はコードが二つ割り当てられることになりました。無視したことが悪いと言っているわけではなく、すでにある文書などとの互換性を考慮した結果、単純に共存させるしかなかったものと考えられますが、このことがいわゆる全角・半角問題を引き起こすことになります。全く同じであるべき英単語が、全角で表現したら検索できないとか、全角・半角を混ぜると、うまく辞書順に並べられないとかの問題です。

一方、Unicode、ISO/IEC 10646 に対しては UTF-8 と呼ばれる encoding がよく用いられています。この encoding は可変長であり、ASCII 集合に対しては従来のコードと全く一致しています。また、開始オクテットやオクテット長が容易に判別できるようになっており、計算機上での処理も負担が少ないのが特徴です。

付 録C　整形入出力

入出力ストリームクラスは整形入出力の際に参照する変数をいくつか持っています。これらはメンバ関数を使って、現在値の取得や変更を行うこともできますし、操作子を演算子 >> や << の間に入れることで変更することもできます。

C.1　ios_base の整形入出力用フラグ

ベースクラスである std::ios_base には std::ios_base::fmtflags 型の整形入出力用変数があります。この変数には表 C.1 に示す要素があり、ビット演算により設定・解除できます。

この変数の現在値は std::ios_base の公開メンバ関数

```
fmtflags flags () const;
```

により取得することができます。入出力ストリームクラスは、std::ios_base から派生してるので、入出力ストリームオブジェクトの関数として呼び出すことができます。この値の変更は公開メンバ関数

```
fmtflags flags ( fmtflags fmtfl );
```

を呼び出すことで可能です。

表 C.1 に示す操作子のうち個別フラグ操作子は先頭に no を付けることにより解除になります。例えば uppercase を解除するには nouppercase とします。

出力フィールド内での位置は adjustfield 操作子で指示します。

数値の表示に関して、basefield 操作子は整数の表示方法を、floatfield 操作子は浮動小数点数の表示方法を指示します。

C.2　入出力ストリームの操作子

入出力ライブラリに含まれる操作子を表 C.2 に示します。

通常、istream のオブジェクトは既定値として skipws flag が設定されています。その場合、ws 操作子を用いなくとも、先行する空白文字は読み飛ばされます。

C.3　標準操作子

操作子のうち、パラメータを持つものは <iomanip> に定義されており、標準操作子（standard manipulators）と呼ばれます。これらは表 C.3 に示してあります。

プログラム C.1 に簡単な使用例を示します。

表 C.1: fmtflags の効果

要素	設定された場合の効果	効果の及ぶストリーム
個別フラグ操作子		
boolalpha	真理値を false か true で表す。	入力・出力
showbase	整数の表示に前に基数を付ける。	出力
showpoint	浮動小数点数を常に小数点付きで表示する。	出力
showpos	非負の数に符号 + を付けて表示する。	出力
skipws	先行する空白文字を読み飛ばす。	入力
unitbuf	出力操作の度にバッファを全部出力する。	出力
uppercase	アルファベット文字の小文字を大文字で表示する。	出力
adjustfield 操作子		
internal	中央詰めで表示する。	出力
left	左詰めで表示する。	出力
right	右詰めで表示する。	出力
basefield 操作子		
dec	整数を 10 進数表示で扱う。	入力・出力
hex	整数を 16 進数表示で扱う。	入力・出力
oct	整数を 8 進数表示で扱う。	入力・出力
floatfield 操作子		
fixed	浮動小数点数の小数点の位置を固定して表示する。	出力
scientific	浮動小数点数を科学計算記法で表示する。	出力

表 C.2: 入出力ストリームの操作子

操作子	効果
入力ストリームの操作子	
ws	可能な限り入力ストリームから空白文字を読み飛ばす。
出力ストリームの操作子	
endl	改行文字を挿入した後、flush する。
ends	null 文字を挿入する。
flush	バッファにある文字すべてを出力装置へ送り出す。

表 C.3: 標準操作子

操作子	効果
`setiosflags(ios_base::fmtflags mask)`	`fmtflags` の各要素を設定する。
`resetiosflags(ios_base::fmtflags mask)`	`fmtflags` の各要素を解除する。
`setbase(int base)`	basefield 操作子を設定する。 `base` は 8,10,16 のいずれか。
`setfill(char c)`	埋め込み文字を `c` に設定する。
`setprecision(int n)`	浮動小数点数の出力の 10 進精度を設定する。
`setw(int n)`	次の挿入に対するフィールド幅を設定する。

プログラム C.1: 標準操作子使用例

```
1  #include <iostream>
2  #include <iomanip>
3  int main()
4  {
5          int x = 17;
6          std::cout
7                  << std::setfill('=')
8                  << std::setw(10)
9                  << x
10                 << std::endl;
11         return 0;
12 }
```

このプログラムの実行結果は

```
========17
```

となります。

付 録D　演算子の優先順位と結合則

式の中に現れる演算子は一つとは限りません。演算子が複数現れた場合には、その演算順序が問題になります。同一の演算子が複数並んだ場合の演算順序の規則を結合則と言います。また異なる演算子が並んだときの演算順序は優先順位が決まっています。ここでは、これらの規則について述べます。

表 D.1 は、組み込みの演算子を優先順位の順に並べたものです。水平線で区切られたブロックの中は同じ優先度で、上（最初）のブロックほど優先度が高くなります。

D.1　オペランドの評価順序

一般にオペランドの評価順序は決まっていません。例えば式 `f() + g()` を評価するとき、`f()` と `g()` のどちらが先に評価されるかはわかりません。

ただし、次の二項演算子

```
op1 && op2
op1 || op2
op1 , op2
```

および三項演算子

```
op1 ? op2 : op3
```

では第一オペランドが最初に評価されます。

特にこのうち論理演算 `&&` と `||` については、第一オペランドを評価した時点で式の結果が確定できるならば、第二オペランドは評価されません（short-circuit evaluation）。従って例えば

```
int x, y, z;
        :
if ((x != 0) && ((y / x) > z))
        :
```

の `if` 文では、`x` が 0 の時、除算 `y / x` は実行されません。つまり、ゼロ除算に対し防御されています。

増分`++` や減分`--` は評価時点でオブジェクトの値が変わりますので、一つの式の中に同一オブジェクトに対する増分や減分が複数表れると多くの場合、結果は不定になることに注意します。例えば

```
int i = 1;
i = ++i - ++i;
```

は減算の右オペランドが先に評価されるのか、左オペランドが先に評価されるのかで結果が変わります。

D.2 結合則

一つの式の中に同一演算子が並べられているときの演算順序を決めるのが結合則（associativity）です。たとえば加算演算子 + では

```
a + b + c
```

は、左の + から先に評価されます。つまり

```
(a + b) + c
```

と同じです。これを表 D.1 では、「左から右」と表しています。

逆に、代入演算子 = の結合則は「右から左」です。

```
a = b = c = 0
```

は

```
(a = (b = (c = 0)))
```

と同じですので、a、b、c いずれも 0 になります。

D.3 優先順位

一つの式の中にある異なる演算子には優先順位があります。優先順位はグループで分けられていて、同じグループに属する演算子は、そのグループの結合則に従います。表 D.1 では、水平線によりグループの堺を示しています。例えば加算 + と減算 - は同じグループですので、優先順位は同じなので、結合則に従います。従って

```
a + b - c
```

は、左の + から先に評価されます。つまり

```
(a + b) - c
```

と同じです。しかし、乗算 * は加算より優先度が高いので、

```
a + b * c
```

は、乗算 * から先に評価されます。つまり

```
a + (b * c)
```

と同じです。ただし、D.1 に述べたように、オペランドの評価順は決まっていません。従って

```
f() + g() * h()
```

は、乗算 * から先に評価され

```
f() + (g() * h())
```

と同じですが、各オペランドの f()、g() および h() のどれがどの順に評価されるかは決まっていません。

表 D.1: 優先順位による組み込み演算子一覧

演算子	役割	式の例	結合則
::	スコープ解決	::x、std::cout	左から右
[]	添字	x[i]	左から右
()	関数呼び出し	f(x, y, z)	左から右
()	型変換	double(i)	左から右
<>()	型変換	static_cast<int>(x)	左から右
typeid	型取得	typeid(obj)	左から右
.	メンバアクセス	x.y	左から右
->	メンバアクセス	p->y	左から右
++、--	後置の増分、後置の減分	x++、x--	左から右
++、--	前置の増分、前置の減分	++x、--x	右から左
~	補数	~x	右から左
!	論理否定	!x	右から左
+、-	正符号、負符号	+x、-x	右から左
*	ポインタ参照	*p	右から左
&	アドレス取得	&x	右から左
sizeof	サイズ取得	sizeof x、sizeof(int)	右から左
new	オブジェクト生成	new double、new char [100]	右から左
delete	オブジェクト解放	delete x、delete [] p	右から左
()	型変換	(int)x	右から左
.*	メンバへのポインタ	x.*y	左から右
->*	メンバへのポインタ	p->*y	左から右
*、/、%	乗算、除算、剰余算	x * y、x / y、x % y	左から右
+、-	加算、減算	x + y、x - y	左から右
<<、>>	左シフト、右シフト	x << n、x >> n	左から右
<、>	値比較	x < y、x > y	左から右
<=、>=	値比較	x <= y、x >= y	左から右
==、!=	等価、非等価	x == y、x != y	左から右
&	ビット単位の論理積	x & y	左から右
^	ビット単位の排他的論理和	x ^ y	左から右
|	ビット単位の論理和	x | y	左から右
&&	論理積	x && y	左から右
||	論理和	x || y	左から右
? :	条件	x ? a : b	右から左
=	（単純）代入	x = y	右から左
+=、-=	（複合）代入	x += y	右から左
*=、/=、%=		x *= y	
<<=、>>=		x <<= n	
&=、^=、|=		x &= y	
throw	例外送出	throw、thorw e	右から左
,	コンマ	x, y	左から右

付 録E　Endian

Endian というのは、数値表現が複数のアドレスを占める時、アドレス順に上位桁から配置するか、下位桁から配置するかという問題です。これは CPU や OS に依存します。アドレス順に上位桁から配置する方式（つまりアドレスの小さい方が上位桁）を big endian、逆にアドレス順に下位桁から配置する方式を little endian と呼びます。

異なる OS 間で数値をバイナリ表現で交換する場合に問題になります。

E.1　ネットワークバイト順

インターネットでは、ネットワークバイト順（network byte order）を定め、この問題に対処しています。インターネットでは big endian が用いられます。この時のアドレスとは早く到達した方が小さい（若い）アドレスとされます。

関連図書

[1] B.W. カーニハン/D.M. リッチー著、石田晴久訳、
”プログラミング言語 C 第 2 版”
ISBN4-320-02692-6、共立出版株式会社

[2] Bjarne Stroustrup 著、株式会社ロングテール/長尾高弘訳、
”プログラミング言語 C++ 第 3 版”
ISBN4-7561-1895-X C3004、アジソン・ウェスレイ・パブリッシャーズ・ジャパン株式会社

[3] Ray Lischner 著、株式会社クイープ訳、
”C++ランゲージ　クイックリファレンス”
ISBN4-87311-191-9 C3055、株式会社オライリー・ジャパン

[4] 柴田望洋 著
”新版明解 C++ 入門編”
ISBN978-4-7973-5454-6 C0055、ソフトバンククリエイティブ株式会社

索 引