Robot Programming II Documentation

リリース *1.0*

NAGASAKA Yasunori

2020 年 09 月 17 日

Contents:

第 1 章

はじめに

1.1 講義で取り上げる内容、目標

ロボットプログラミング II では、C++ 言語の文法、機能を学び、C++ の特徴を活かしたプログラムを作成できるようになることを目指す。そのために、

- 講義資料 (この文書) は、講義の進行に合わせて一通り読んで、内容を理解するように努める。
- サンプルプログラムがある場合は、そこで扱う内容の例となっているので一通り読んで理解するように努める。
- 実行してプログラムの動作や結果を確認できる例では、自分で実際に実行してみる。

を続けてほしい。

C++ は C 言語に **オブジェクト指向** の考え方、表現法を取り入れた言語である。C 言語の上位互換となっているので、C 言語のプログラムは若干の変更で C++ のプログラムとしてコンパイル、実行できる。

この後のロボットプログラミング II の講義では主要な内容として、

- コンパイラの機能と使い方
- デバッグの方法、デバッガの使い方
- C, C++, プログラム作成の基本スキル
- C++ の様々な機能
- オブジェクト指向 1、class
- オブジェクト指向 2、継承
- 標準テンプレートライブラリ, STL : Standard Template Library
- 総合演習課題

を取り上げる。

教科書は指定しない。以下にいくつか参考文献、資料を挙げておく。ここに挙げたもの以外でも、多くの書籍、公開されている資料があるので、自分に合っていると思うものを入手して講義の前後で必要に応じて参照するとよい。

1.1.1 参考文献

- **「ロベールの C++ 入門講座」**, **ロベール**, **マイナビ**, ISBN-13: 978-4-8399-2605-2, 2007 **年** 11 **月**, (900 ページを超えるので全体を読み通すのは根性がいるが、説明は大変詳しい。)
- **「C++ の絵本 第 2 版」**, **株式会社アンク**, **翔泳社**, ISBN-13: 978-4798151908, 2017 **年** 4 **月**, (C++ の機能に絞って説明しているので、C 言語を事前に理解している人向け。200 ページを少し超える程度なので読みやすい。ページ数が少ない分、説明は簡潔である。)
- **「C++ 入門」**, http://wisdom.sakura.ne.jp/programming/cpp/ (C++ を解説する Web サイトである。印刷すると 110 ページ程度であるが、一通りの機能がしっかり解説されている。)
- **「C 言語経験者のための C++ 入門」**, **金森由博**, http://kanamori.cs.tsukuba.ac.jp/index-ja.html#for_students (28 枚のスライドで C++ の要点が簡潔に説明されている。C 言語を事前に理解している人向け。すべての機能が含まれるわけではないが、基本はここに書かれている内容でほぼカバーされている。)

1.2 実習の環境

プログラミングの実習は次の環境で行う。

- OS は Ubuntu 16.04, または 18.04
- コンパイラは OS に付属する gcc, g++

OS の version は上記のどちらでもよい。サンプルプログラムの動作確認は 18.04 上で行っている。16.04 では付属するコンパイラの version の差異により、稀ではあるがコンパイル時の挙動や結果に違いが生じる可能性がある (過去に一度あった) 。明らかに正しいプログラムで問題が生じる場合は担当教員に問い合わせる。

この後の説明において実行するコマンドは次のように表記する。例としてファイルの一覧を表示するコマンド ls では、

```
$ ls -l
```

行頭の $ は、ターミナル上のプロンプト (次にコマンドを入力する場所) を表す。

プログラム作成時に使用するエディタは vi, vim, nano, gedit, emacs など、使い慣れているものでよい。担当教員は emacs を使用するがそれに合わせる必要はない。例として、編集したいファイルを sample.c とすると、

```
$ vim sample.c
```

サンプルプログラムをまとめて圧縮したファイル rp2src.tgz を以下の方法、場所で公開する。内容は同じであるので、どれかを選んで最初の講義の前に自分の PC にダウンロード、展開して参照できるようにしておく。

- 講義資料を置く Web サイト、http://edu.isc.chubu.ac.jp/naga/
- Google Classroom のこの講義のクラス
- メールに添付して送る。

拡張子 tgz のファイルをダウンロードした後の展開方法は次になる。

- GUI 環境ではダブルクリックして展開する。
- CUI 環境では次のコマンドで展開できる。ファイル名を file.tgz とすると

```
$ tar zxvf ./file.tgz
```

1.3 Linux の基本コマンドの理解度の確認

講義では Linux の端末、ターミナルソフト上でプログラムの編集やコンパイル、実行を行う。その際に Linux のファイル操作に関する基本コマンドの操作法を理解している必要がある。

★ 練習

以下の練習問題に記されている操作をするためのコマンドを考えてみよう。操作の内容によっては複数のコマンドを組み合わせてもよい。

前提として、

- 現在はホームディレクトリ /home/er14000 にいて作業をしている。
- テキストファイル a, b がある。ディレクトリ d1 がある。

という状況にある。

1.4 C 言語の復習

この講義では、C 言語の基礎は理解しているという前提で、プログラミングや C++ に関する講義や実習を進める。

★ 練習

C 言語の復習のための課題を章末の練習問題として示す。それらの課題を考える過程で、理解が不十分な点があれば早めに復習しておいてほしい。

1.5 まとめ

この章の目標

- この講義で扱う内容が理解できる。
- 参考文献から自分にあったものを選ぶことができる。
- Linux の基本コマンドの使い方がわかる。
- C 言語の基礎を理解して簡単なプログラムが作成できる。

練習問題

1. C++ 言語の特徴を簡単に説明しなさい。

Linux の基本コマンド

2. ディレクトリ d2 を作る。
3. ファイルの一覧を表示する。
4. ファイルの一覧を詳細情報とともに表示する。
5. ファイルの一覧を表示するコマンドをオンラインマニュアルで調べる。
6. d2 に移動する。
7. 現在いるディレクトリを表示する。
8. 一つ上の階層のディレクトリに移動する。
9. d1 に移動する。
10. ホームディレクトリに移動する。
11. a の内容を表示する。
12. a の名前を c に変更する。
13. b を e という名前で複製する。
14. e を d1 に移動させる。
15. b, c を d2 に移動させる。
16. c を削除する。
17. b を d1 に移動させる。
18. d2 を削除する。
19. d1 を削除する。

C 言語の基礎の確認

20. KB(Keyboard) から int n を入力して 1+2+3+...+n を求める。(e1-9.c)
21. KB から int n を入力して奇数か偶数か判定する。(e1-10.c)
22. KB から 文字列を入力して、何文字あるか数える。(e1-11.c)
23. KB から int a, b を入力したら a の b 乗を求める関数 int f(int a, int b) を作る。計算結果は返り値として呼び出し元に返す。(e1-12.c)

第 2 章

コンパイル、実行ファイルの生成

2.1 C と C++ の違い

2.1.1 C++ は C 言語の上位互換、拡張子が異なる

C++ 言語は C 言語の上位互換で、C 言語の機能に加えて多くの機能が拡張されている。よって C 言語で書かれたプログラムは若干の変更で C++ のプログラムに変更することができる。ソースファイルの拡張子は C 言語が .c なのに対して、C++ では .cpp .c++ .cxx など複数種類が認められている。一般的に使われるのが .cpp であるのでこの講義では .cpp を採用する。

C 言語のプログラムの例を次に示す。

リスト 2.1　2-1.c

```
1  // gcc -Wall -ansi -std=c99 -o 2-1 2-1.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7      printf("Hello world. (2-1.c)\n");
8
9      exit(EXIT_SUCCESS);
10 //   exit(EXIT_FAILURE);
11 }
```

stdio.h は printf() の定義、stdlib.h は exit() の定義で必要とされる。exit() は返り値を返すとともにプログラム終了時に必要な処理を行う。return() を指定してもよい。その場合、プログラム終了時に必要な処理は OS が行う。返り値の EXIT_SUCCESS は実行が成功した場合、または正常終了を表す値で実体は整数の 0 が定義されている。コメントにしてある EXIT_FAILURE は実行が失敗した場合、または異常終了を表す値で実体は整数の 1 が定義されている。exit() を使うとそこで実行が終了することが視覚的にわかる。return を使用しても問題ない。

コンパイル時は以下のように gcc を指定する。これは最も単純なコンパイルの指定で、実際は 2-1.c の 1 行目のコ

メントのようにオプションをいくつか指定する (説明は後出) 場合が多い。 ./a.out で実行できる。

★ 練習

コンパイルと実行を試してみよう。

```
$ gcc 2-1.c
$ ./a.out
```

2.1.2 ヘッダの指定を変えればコンパイル可能

C++ 言語のプログラムの例を次に示す。2-1.c のヘッダの指定を変更して、拡張子を.cpp に変更したものである。

リスト 2.2　2-2.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 2-2 2-2.cpp
2  #include <cstdio>
3  #include <cstdlib>
4
5  int main(void)
6  {
7      printf("Hello world. (2-2.cpp)\n");
8
9      exit(EXIT_SUCCESS);
10 //  exit(EXIT_FAILURE);
11 }
```

コンパイル時は以下のように g++ を指定する。こちらも実際はオプションをいくつか指定する (後出) 場合が多い。

```
$ g++ 2-2.cpp
$ ./a.out
```

このようにもとが C 言語のプログラムであっても、

- ヘッダの指定を変更する。 stdio.h → cstdio
- 拡張子を .cpp に変更する。

ことで、C++ の機能を内部で全く使っていなくても C++ のプログラムとしてコンパイル、実行することができる。

C++ 言語の別のプログラムを次に示す。C++ 固有の記述があるので C 言語とは互換性がない。

リスト 2.3　2-3.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 2-3 2-3.cpp
2  #include <cstdio>
3  #include <cstdlib>
4
```

(次のページに続く)

(前のページからの続き)

```
class a {
    int a;
} b;

int main(void)
{
    a c;

    printf("Hello world. (2-3.cpp)\n");

    exit(EXIT_SUCCESS);
//  exit(EXIT_FAILURE);
}
```

```
$ g++ 2-3.cpp
$ ./a.out
```

gcc には .c 、g++ には .cpp を指定するのが本来の使い方であるが、では gcc に .cpp 、g++ には .c を指定したらどうなるか試してみよう。

```
$ g++ 2-1.c
$ gcc 2-2.cpp
$ gcc 2-3.cpp
```

拡張子とプログラムの内容が一致していればエラーにならないことがわかる。これは gcc と g++ は実体が同じであるので、内部で適切な処理に切り替えていると推測できる。

以下では拡張子だけを変更してコピー、コンパイルしてみる。これをすると、拡張子が .c であるが内容に C++ の記述が含まれるプログラムになる。

```
$ cp 2-1.c 2-1.cpp
$ gcc 2-1.cpp
$ g++ 2-1.cpp
```

```
$ cp 2-2.cpp 2-2.c
$ gcc 2-2.c
$ g++ 2-2.c
```

```
$ cp 2-3.cpp 2-3.c
$ gcc 2-3.c
$ g++ 2-3.c
```

この場合はある組み合わせの場合にエラーとなる。gcc に拡張子 .c のファイルを指定するが、内容に C++ の記述が含まれるとコンパイルができない。

ここでは、コンパイラ、拡張子、プログラムの内容の組み合わせを様々に組み合わせてその時のコンパイラの動作

を確認したが、これはあくまでも実験である。通常このような組み合わせを指定すると、うまく動くこともあるが、トラブルのもとになるので、gcc には C 言語のプログラム、g++ には C++ のプログラムを指定すべきである。

2.1.3 C++ ではライブラリの指定が不要

C 言語では、言語の仕様でもともと提供されていない機能を使いたい場合は、その機能の実行可能な機械語プログラムをひとまとまりのファイルにパックした **ライブラリ** というファイルをコンパイル時に指定して実行ファイルに組み込む必要がある。

次のプログラムでは三角関数 sin を計算する数学関数 sin() を利用している。

リスト 2.4　2-4.c

```
// gcc -Wall -ansi -std=c99 -o 2-4 2-4.c -lm
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
    printf("sin(1.5708) = %f (2-4.c)\n", sin(1.5708));
    printf("sin(3.1416) = %f\n", sin(3.1416));

    exit(EXIT_SUCCESS);
}
```

★ 練習

試してみよう。

```
$ gcc 2-4.c -lm
$ ./a.out
```

関数 sin() を使うには次の 2 つの指示が必要となる。

- インクルードするヘッダとして math.h を指定する。
- 関数の実体を含むライブラリ libm をコンパイル時に -lm として指定する。

math.h の実体は /usr/include/math.h にある。stdio.h, stdlib.h の実体もここにある。libm の実体は /lib/x86_64-linux-gnu/libm.so.6 にある。

関数名がわかれば **オンラインマニュアル** で入出力の仕様、ヘッダ、ライブラリを調べることができる。man コマンドは上下の矢印キーで 1 行ずつスクロール、スペースキーで 1 ページスクロール、'q' で終了する。

★ 練習

試してみよう。

```
$ man sin
$ man printf
$ man scanf
```

C++ では必要なライブラリをコンパイラが管理してくれるのでユーザが指定しなくてもよい。

リスト 2.5　2-5.cpp

```
// g++ -Wall -ansi -std=c++11 -o 2-5 2-5.cpp
#include <cstdio>
#include <cstdlib>
#include <cmath>

int main(void)
{
    printf("sin(1.5708) = %f (2-5.cpp)\n", sin(1.5708));
    printf("sin(3.1416) = %f\n", sin(3.1416));

    exit(EXIT_SUCCESS);
}
```

★ 練習

試してみよう。

```
$ g++ 2-5.cpp
$ ./a.out
```

2.2 コンパイラの使い方、オプション

gcc, g++ にはコンパイラの動作を指示するための様々なオプションが用意されている。ここではよく使われる代表的なオプションの機能を理解する。

代表的なオプションの機能

'-c'　コンパイルはするが **リンク** は行わず、オブジェクトファイルを出力する。分割コンパイルを行う場合に使われる。

'-o'　出力ファイルの名前を指定する。指定しないと実行ファイルは a.out になる。'-c' を指定して出力が実行ファイルでない場合はソースファイルの拡張子を'.o' に変えたファイル名になる。

'-O'　実行ファイルの速度やサイズを改善するように機械語命令の構成を **最適化** する。最適化のレベルや対象を以下のように指定することができる。

```
'-O0' : （ラージオーゼロ） コンパイル時間を短くする。速度の改善はしない。デフォルト。
'-O' '-O1' : 速度を改善する。コンパイル時間は長くなり、メモリ使用量が増える。
```

(次のページに続く)

(前のページからの続き)

```
'-O2' : 更に速度を改善する。コンパイル時間は長くなり、メモリ使用量が増える。
'-O3' : 更に速度を改善する。コンパイル時間は長くなり、メモリ使用量が増える。
'-Os' : 実行ファイルのサイズを小さくする。
```

'-ggdb' **デバッガ** gdb 用の情報を実行ファイルに付加する。

'-I' インクルードするヘッダが置かれている標準のディレクトリ以外の path を指定する。

'-L' リンクするライブラリが置かれている標準のディレクトリ以外の path を指定する。

'-l' リンクするライブラリを指定する。

'-Wall' 主要な警告を表示する。

'-ansi' ANSI で定められた規格を適用する。

'-std=c99' 1999 年に定められた C 言語の規格を適用する。

'-std=c++11' 2011 年に定められた C++ の規格を適用する。

gcc, g++ にどのようなオプションがあるか、詳細はオンラインマニュアルで調べることができる。

```
$ man gcc
$ man g++
```

gcc, g++ を使用するときは以下のオプションを指定するとよい。実行速度を改善したい場合、デバッガを使用する場合はそれぞれのオプションを追加する。

★ 練習

試してみよう。

```
$ gcc -Wall -ansi -std=c99 -o 2-1 2-1.c
$ g++ -Wall -ansi -std=c++11 -o 2-2 2-2.cpp
```

2.2.1 '-o' オプション

これまでのコンパイルの指定の仕方では実行ファイルの名前はいつも a.out になる。

```
$ gcc 2-1.c
$ ./a.out
```

'-o' オプションを使うと実行ファイルの名前を指定できる。

```
$ gcc -o 2-1 2-1.c
$ ./2-1
```

2.2.2 '-O' オプション

以下のプログラムは sin(), cos() を組合わせて求めた値を配列の要素として代入していき、最後にその和を求める。最適化オプションを指定することで実行速度がどのように変化するか比較してみる。コマンド time を指定することで実行にかかった時間を計測できる。

リスト 2.6　2-6.c

```
// gcc -Wall -ansi -std=c99 -o 2-6 2-6.c -lm
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define  SIZE 512

int main(void)
{
  int i, j, k;
    double a[SIZE][SIZE], sum = 0.0;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) a[i][j] = 0.0;
    }
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
              for (k = 0; k < 256; k++) {
                a[i][j] = sin(i + j) + cos(i - j) - sin(a[i][j]) - cos(a[i][j]);
            }
        }
    }
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) sum += a[i][j];
    }
    printf("%f\n", sum);

    exit(EXIT_SUCCESS);
}
```

★ 練習

試してみよう。

```
$ gcc -o 2-6 2-6.c -lm
$ time ./2-6
```

```
$ gcc -O0 -o 2-6 2-6.c -lm
$ time ./2-6
```

```
$ gcc -O -o 2-6 2-6.c -lm
$ time ./2-6
```

```
$ gcc -O2 -o 2-6 2-6.c -lm
$ time ./2-6
```

```
$ gcc -O3 -o 2-6 2-6.c -lm
$ time ./2-6
```

次に、2-6.c の最後の printf() をコメントアウトして、'-O' を指定した場合、指定しない場合の実行速度を比較してみる。

```
$ gcc -o 2-6 2-6.c -lm
$ time ./2-6
```

```
$ gcc -O -o 2-6 2-6.c -lm
$ time ./2-6
```

printf() の有無で実行速度の変化の度合いが変わるのが見てとれるが、なぜこのような違いが現れるのだろう？

2.2.3 '-Wall' オプション

次のプログラムは scanf() printf() を使った簡単な入出力のプログラムである。警告を表示する '-Wall' オプションの有無でコンパイル時の表示がどのように変わるか比較してみる。

リスト 2.7　2-7.c

```
1  // gcc -Wall -ansi -std=c99 -o 2-7 2-7.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7      int i, j;
8
9      printf("Input number => ");
10     scanf("%d", &i);
11     if (i = 1) printf("%d\n", i);
12
13     exit(EXIT_SUCCESS);
14 }
```

★ 練習

試してみよう。

```
$ gcc -o 2-7 2-7.c
$ ./2-7
```

```
$ gcc -Wall -o 2-7 2-7.c
$ ./2-7
```

文法的に間違いがあるとエラーとなり、コンパイルは中断され実行ファイルは生成されない。'-Wall' オプションで表示される警告は、文法的には間違いではないが、意味のない記述、間違いや勘違いの可能性がある部分である。警告は無視しても、実行ファイルは生成されており実行もできるので、指摘された部分に問題がないことを理解していれば放置することもできる。しかし問題となる可能性がある部分なので、メッセージの内容を一通り確認しておくのがよい。

2.3 分割コンパイル、リンク

プログラムの規模が大きくなると、プログラム全体を単一のソースファイルで作成すると様々な不都合や問題が生じる。例えば数千行に及ぶようなソースファイルでは、どこに何が書いてあるか理解しておくことは難しくなる。ある機能を改良したいと思ってソースコードを見ようと思っても目的の場所までスクロールさせるのに手間がかかる。プログラムの規模が大きくなるとコンパイルの時間も比例して長くなるので、1 箇所変更しただけでもその結果を確認するためのコンパイルが終わるまで長時間待たなくてはならない。大規模なプログラムではコンパイルに数時間かかることもある。またソースファイルが単一の場合、複数人のチームで協力して開発するのが難しくなる。複数の人が並行して別々の場所を編集していると、その結果を矛盾なく統合することが難しくなる。

このようにソースファイルが長くなると生じる不都合を回避するために、**分割コンパイル**という手法が用いられる。分割コンパイルとは、プログラムの持つ多くの機能毎にソースファイルを分割して、それぞれを独立して開発できるようにする。

次のプログラムはいくつかの定義や関数を含むプログラムである。プログラムの動作と結果には特に意味はない。

リスト 2.8　4-6.c

```
1  // gcc -Wall -ansi -std=c99 -o 4-6 4-6.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define        A        100
6  #define        B        10
7  #define C         (A+B)
8
9  void        f1(int a[]);
10 void        f2(int b[]);
11 void        f3(int a[], int b[], int c[]);
12
13 char        name[] = "abc";
14
```

(次のページに続く)

(前のページからの続き)

```
int main(void)
{
    int         a[A], b[B], c[C];

    f3(a, b, c);

    exit(EXIT_SUCCESS);
}

void        f1(int a[])
{
    for (int i = 0; i < A; i++) a[i] = i;;
}

void        f2(int b[])
{
    for (int i = 0; i < B; i++) b[i] = i;
}

void        f3(int a[], int b[], int c[])
{
    int sum = 0;

    printf("%s\n", name);
    f1(a);
    f2(b);
    for (int i = 0; i < A; i++) c[i] = a[i];
    for (int i = 0; i < B; i++) c[i + A] = b[i];
    for (int i = 0; i < C; i++) sum += c[i];
    printf("%d\n", sum);
}
```

2.3.1 分割コンパイルの実際

上記のプログラムを複数のファイルに分割して、個々の機能を独立して変更できるようにすると次のようになる。分割コンパイルの考え方を示すために、個々のファイルは意図的に短くして、ファイル数が多くしてある。

ヘッダファイル (.h) には定義や宣言を記述する。関数のプロトタイプ宣言なども書けるが、関数の実体 (実行する内容) は書かない。複数のソースファイルで必要になるような内容をヘッダにまとめて書く。

リスト 2.9　4-7a.h

```
#ifndef         HEADER_47A
#define         HEADER_47A

#define         A         100
```

(次のページに続く)

(前のページからの続き)

```
5
6  #endif // HEADER_47A
```

リスト 2.10　4-7b.h

```
1  #ifndef HEADER_47B
2  #define        HEADER_47B
3
4  #define        B        10
5
6  #endif // HEADER_47B
```

リスト 2.11　4-7c.h

```
1  #ifndef HEADER_47C
2  #define HEADER_47C
3
4  #include "4-7a.h"
5  #include "4-7b.h"
6
7  #define C        (A+B)
8
9  #endif // HEADER_47C
```

ソースファイル (.c) には、関数の実体を書く。定義や宣言はそのファイルでだけ必要なものであればソースファイルに書いてもよい。ヘッダにすべて書くようにしてもよい。

リスト 2.12　4-7main.c

```
1  // gcc -Wall -ansi -std=c99 -c 4-7main.c
2  // gcc -Wall -ansi -std=c99 -o 4-7 4-7main.o 4-7a.o 4-7b.o 4-7c.o
3  // gcc -Wall -ansi -std=c99 -o 4-7 4-7main.c 4-7a.c 4-7b.c 4-7c.c
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "4-7a.h"
8  #include "4-7b.h"
9  #include "4-7c.h"
10
11 void        f3(int a[], int b[], int c[]);
12
13 char        name[] = "abc";
14
15 int main(void)
16 {
17     int        a[A], b[B], c[C];
18
19     f3(a, b, c);
```

(次のページに続く)

(前のページからの続き)

```
20
21      exit(EXIT_SUCCESS);
22  }
```

リスト 2.13　4-7a.c

```
1   // gcc -Wall -ansi -std=c99 -c 4-7a.c
2   #include "4-7a.h"
3
4   void        f1(int a[])
5   {
6       for (int i = 0; i < A; i++) a[i] = i;;
7   }
```

リスト 2.14　4-7b.c

```
1   // gcc -Wall -ansi -std=c99 -c 4-7b.c
2   #include "4-7b.h"
3
4   void        f2(int b[])
5   {
6       for (int i = 0; i < B; i++) b[i] = i;
7   }
```

リスト 2.15　4-7c.c

```
1   // gcc -Wall -ansi -std=c99 -c 4-7c.c
2   #include <stdio.h>
3   #include "4-7a.h"
4   #include "4-7b.h"
5   #include "4-7c.h"
6
7   void        f1(int a[]);
8   void        f2(int b[]);
9
10  extern        char name[];
11
12  void        f3(int a[], int b[], int c[])
13  {
14      int sum = 0;
15
16      printf("%s\n", name);
17      f1(a);
18      f2(b);
19      for (int i = 0; i < A; i++) c[i] = a[i];
20      for (int i = 0; i < B; i++) c[i + A] = b[i];
21      for (int i = 0; i < C; i++) sum += c[i];
22      printf("%d\n", sum);
```

(次のページに続く)

(前のページからの続き)

```
23 }
```

4-7main.c の冒頭には、コンパイルの指示が 3 種類示されているが、必要に応じて使い分けるとよい。最初の例では、オプションとして -c が指定されているが、これはコンパイルの結果として実行ファイルではなく **オブジェクトファイル** を出力する。

★ 練習

試してみよう。

```
gcc -c 4-7main.c
```

4-7main.c からはファイル名を指定しなければオブジェクトファイルとして 4-7main.o が出力、記録される。同様に、4-7a.c, 4-7b.c, 4-7c.c についてもオブジェクトファイルを生成できる。オブジェクトファイルに他のオブジェクトファイルやライブラリ、起動時や終了時に必要な処理を追加すると実行ファイルにすることができる。この処理を **リンク** という。次のコマンドでオブジェクトファイル群をリンクして実行ファイル 4-7 を生成することができる。

```
gcc -o 4-7 4-7main.o 4-7a.o 4-7b.o 4-7c.o
```

また次のコマンドではオブジェクトファイルの生成とリンクを一度に行う。この場合、オブジェクトファイルは処理の過程で生成されるが保存はされず、実行ファイルが生成された後は廃棄される。これまで指定していた gcc の指示は、オブジェクトファイルの生成とリンクを連続して行うように指示していたことになる。

```
gcc -o 4-7 4-7main.c 4-7a.c 4-7b.c 4-7c.c
```

ファイルを分割すると、それぞれのファイル間で依存関係が生じるので、コンパイルする時には抜けがないように注意する必要がある。例として、4-7a.h では A が 100 として定義されているが、何らかの理由で 200 に変更したとする。その場合、ヘッダファイルを書き換えただけでは不十分で、4-7a.h をインクルードしているソースファイルをすべて再コンパイルし直さないと変更が反映されない。一部のソースファイルをコンパイルし忘れると、ある部分の処理では A = 100 として扱われ、別の処理では A = 200 となり、不整合が生じることになる。

ソースファイル、ヘッダファイルを分割しても、次のようにすべてインクルードするように書くのはよくない。分割したソースファイルをすべてインクルードしているので、これでは単一のファイルにすべて書いたのと同じである。ファイル毎に独立したオブジェクトファイルは生成されない。どこか 1 箇所でも変更したら、プログラム全体、すべての行を再コンパイルする必要がある。

リスト 2.16　4-8.c

```
1 // gcc -Wall -ansi -std=c99 -o 4-8 4-8.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "4-7a.h"
5 #include "4-7b.h"
```

(次のページに続く)

(前のページからの続き)

```
6  #include "4-7c.h"
7
8  #include "4-7a.c"
9  #include "4-7b.c"
10 #include "4-7c.c"
11
12 int main(void)
13 {
14     int         a[A], b[B], c[C];
15
16     f3(a, b, c);
17
18     exit(EXIT_SUCCESS);
19 }
```

2.3.2 分割コンパイルを可能にするための条件

分割コンパイルを行う際には次の事柄を考える必要がある。

- ファイルを分割するときには、何らかの関数、または関連する関数をまとめた機能を単位として (機能モジュール) 分けるのが一般的であるが、どのように機能を分けるか。
- その機能モジュールへの入力は何か、出力は何か。
- その機能モジュールが必要とする定義や宣言は何か。
- その機能モジュールが必要とするモジュール外部の関数やモジュールは何か。

これらの事柄が明確になると、他の部分とは独立してモジュールのプログラム作成が可能となり、それぞれのモジュールを並行して開発することができる。

2.4 ヘッダに書く内容、二重定義を防ぐ

ヘッダには次の要素を書く。

- インクルードするヘッダ
- #define による名前や定数の定義
- グローバル変数の定義
- 関数のプロトタイプ宣言

上記の要素で、単一のソースファイルでのみ必要とされるものについてはそのファイルに直接書いてもよいが、そのファイル用の専用のヘッダファイルにしてもよい。複数のソースファイルで必要とされる要素は、ヘッダファイ

ルで 1 回のみ、1 箇所で定義して複数のファイルでインクルードする。これにより内容を変更する場合は 1 箇所だけの書き換えでよくなり、保守性が高まり、間違える可能性が低くなる。

ヘッダファイルを複数回インクルードして、定義や宣言が複数回行われると文法エラーや間違いのもととなる。よってヘッダファイルを記述する場合は、複数回インクルードされてもその内容が反映されるのは 1 回だけとなるよう、次のような記述をする。

リスト 2.17　4-7a.h

```
1 #ifndef        HEADER_47A
2 #define        HEADER_47A
3
4 #define        A        100
5
6 #endif // HEADER_47A
```

1, 2, 6 行目が二重定義を防ぐ仕組みになっている。1 行目は「もし HEADER_47A が定義されていなければ (以降を有効にする) 」、2 行目は HEADER_47A を定義する、6 行目は 1 行目からの処理をここで終える、という意味になる。もともとこのヘッダでしたかったのは 4 行目の定義だけである。二重定義を防ぐ仕組みにより、4 行目はこのヘッダが最初にインクルードされたときのみ有効になる。2 回目以降のインクルードでは 1 回目のインクルードで既に HEADER_47A が定義されているので 4 行目はスキップされる。

この方法のポイントは、HEADER_47A のように各ヘッダファイル毎に固有の文字列を使用することである。複数のヘッダファイルで同じ文字列を使ってしまうと、二重定義を防ぐ仕組みが機能しなくなる。

2.5 分割コンパイルを支援するツール make

ヘッダファイルやソースファイルの数が多くなると、それぞれのファイル間の依存関係を人が管理するのが難しくなり、見落としや勘違いが生じる。そのような間違いを防ぐために **make** という分割コンパイルを行う時の開発支援ツールがある。make では、専用の書式でファイル間の依存関係を事前に記述しておくだけで、コンパイル時にはそれらのルールに従って、必要なファイルだけ検出してコンパイルしてくれる。次に示すのは、先に示した 4-7*.c 群の依存関係を記述したファイルである。

リスト 2.18　Makefile (Standard)

```
1  4-7: 4-7main.o 4-7a.o 4-7b.o 4-7c.o
2          gcc -o 4-7 4-7main.o 4-7a.o 4-7b.o 4-7c.o
3
4  4-7main.o: 4-7main.c 4-7a.h 4-7b.h 4-7c.h
5          gcc -c 4-7main.c
6
7  4-7a.o: 4-7a.c 4-7a.h
8          gcc -c 4-7a.c
9
10 4-7b.o: 4-7b.c 4-7b.h
```

(次のページに続く)

(前のページからの続き)

```
11          gcc -c 4-7b.c
12
13  4-7c.o: 4-7c.c 4-7a.h 4-7b.h 4-7c.h
14          gcc -c 4-7c.c
```

ここに記述されているのは、２行ずつが単位となっていて、ターゲットを生成するにはどのファイルが必要とされて、コマンドとして何を実行すればよいかが記述されている。ファイル名は Makefile とするのが一般的で、この名前にすればファイル名の指定を省略できる。

ターゲットの生成ルールの一般形は次のようになる。コマンド行を指定する行は先頭に TAB 文字を挿入する必要がある。

生成するファイル：生成するときに必要となるファイル群
**　　生成するためのコマンド行**

Makefile を作成した後で make を実行する時は次のように指定する。

★ 練習

試してみよう。

```
$ make
```

Makrfile 中の記述は先頭から評価、実行されていくので最終的に生成したいターゲットを先に書く。その後、先に指定したターゲットを構成するオブジェクトファイルなどを順に指定する。

make はコンパイルに関する基本的な依存関係、ルールを内蔵しているので、その仕組みを利用すると、Makefile の記述を簡潔にできる。例として、ターゲットが a.o となっている場合は、ソースファイルは a.c がもとになっていると仮定して処理を進める。またその生成ルールは cc -c a.c であるということも組み込まれている。上記の Makefile は以下のように書いても同じように動作する。

リスト 2.19　Makefile (Simple version)

```
4-7: 4-7main.o 4-7a.o 4-7b.o 4-7c.o
        gcc -o 4-7 4-7main.o 4-7a.o 4-7b.o 4-7c.o

4-7main.o: 4-7a.h 4-7b.h 4-7c.h
4-7a.o: 4-7a.h
4-7b.o: 4-7b.h
4-7c.o: 4-7a.h 4-7b.h 4-7c.h
```

2.6 まとめ

この章の目標

- C 言語、C++ それぞれのコンパイラの基本的な使い方がわかる。
- ヘッダとライブラリの指定の仕方がわかる。
- オンラインマニュアルの使い方がわかる。
- 主要なオプションの機能を理解して適切なものを指定できる。
- 分割コンパイル、リンクの考え方、仕組みを理解する
- ヘッダに書く内容にどのようなものがあるか理解する
- make の仕組み、使い方を理解して活用できる

練習問題

1. /usr/include にどのようなヘッダファイルがあるか調べる。
2. /lib/x86_64-linux-gnu/ にどのようなライブラリファイルがあるか調べる。
3. /usr/include/math.h の中で円周率 π (M_PI) の値がどう定義されているか調べる (770 行付近)。
4. /usr/include/math.h の中で自然対数の底 e (M_E) の値がどう定義されているか調べる (770 行付近)。
5. /usr/include/stdlib.h の中で EXIT_SUCCESS の定義されている場所を調べる (90 行付近)。
6. 自然対数の関数 log() の仕様、必要なヘッダ、ライブラリをオンラインマニュアルで調べる。
7. 指数関数 exp() の仕様、必要なヘッダ、ライブラリをオンラインマニュアルで調べる。
8. 2-6.c で最後の printf() をコメントアウトしたときの実行速度の変化の理由を考える。
9. 分割コンパイルを活用するとどのようなメリットがあるか説明しなさい。
10. リンクとは何をすることか説明しなさい。

11. make を使うとどのようなメリットがあるか説明しなさい。

第 3 章

デバッグの方法、デバッガの使い方

プログラムは「作る」能力だけではなく、「間違いを見つける」、「直す」能力も同じように必要とされる。プログラムに含まれる間違いを **バグ** と言い、その間違いを修正する作業を **デバッグ** という。

正しく動かないプログラムは、最後は自分の力で間違いを見つけて修正する必要がある。間違いがあるときに、いつも都合よく誰かに間違いを見つけてもらったり、直してもらえるわけではないので、自力で対処できるように訓練する必要がある。

プログラムを作成する過程では、様々な書き間違い、文法的な間違い、勘違い、文法的には正しいが結果が正しくないロジックの間違い、などで正しく動作するプログラムを完成させるまでには何度も修正と動作確認を行うことがよくある。この章では文法エラーがある場合と、ない場合のそれぞれでどのようにプログラムを直すか、その考え方を理解して実践できるようになることを目指す。

3.1 文法エラーがある場合のデバッグ

3.1.1 C 言語　文法エラーがある場合

文法エラーがある場合は、コンパイルの途中でエラーメッセージが表示されコンパイルは中断される。エラーメッセージには通常プログラムのどこ（何行目）に間違いがあるか示されるので、その内容をもとに間違いを見つけて修正する。エラーメッセージが数多く表示される場合は、先に表示されるものから順に対処する。エラーが非常に多く見える場合でも、運が良ければ最初のエラーを直したところでエラーがなくなることもある。

以下に示すプログラムはそれぞれどこかに間違いがあり、コンパイルできない。プログラムを作成していると、それぞれ日常的によく見かける間違いであるので、間違いの箇所を特定して正しく動くように修正する。

一般的に他人が作成したプログラムを理解して修正することは難しい作業とされているが、以下のプログラムは数行の短いプログラムであるので、挑戦してみよう。次の示す順番で直すことを試みる。

1. ソースファイルを見るだけで間違いの場所を推定する。
2. コンパイルしてみて、エラーメッセージを参考にして間違いを見つける。

3. あらゆる手段を講じて間違いを見つけて修正する。

★ 練習

それぞれのプログラムの間違いを上記の手順で見つけてみよう。

リスト 3.1　3-1.c

```
// gcc -Wall -ansi -std=c99 -o 3-1 3-1.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("Hello world.\n")

    exit(EXIT_SUCCESS);
}
```

リスト 3.2　3-2.c

```
// gcc -Wall -ansi -std=c99 -o 3-2 3-2.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    qrintf("Hello world.\n");

    exit(EXIT_SUCCESS);
}
```

リスト 3.3　3-3.c

```
// gcc -Wall -ansi -std=c99 -o 3-3 3-3.c
#include <stdi0.h>
#include <stdlib.h>

int main(void)
{
    printf("Hello world.\n");

    exit(EXIT_SUCCESS);
}
```

リスト 3.4　3-4.c

```
// gcc -Wall -ansi -std=c99 -o 3-4 3-4.c
#include <stdio.h>
#include <stdlib.h>
```

(次のページに続く)

(前のページからの続き)

```
4
5  int main(void)
6  {
7      printf("Hello world.\n");
8
9      exit(EXlT_SUCCE55);
10 }
```

リスト 3.5　3-5.c

```
1  // gcc -Wall -ansi -std=c99 -o 3-5 3-5.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7      printf("Hello world.\n);
8
9      exit(EXIT_SUCCESS);
10 }
```

リスト 3.6　3-6.c

```
1  // gcc -Wall -ansi -std=c99 -o 3-6 3-6.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7      printf("Hello world.\n");  // １行コメントはこのように書く
8
9      exit(EXIT_SUCCESS);
10 }
```

リスト 3.7　3-7.c

```
1  // gcc -Wall -ansi -std=c99 -o 3-7 3-7.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7      int a;
8
9      printf("Input number -> ");
10     scanf("%d", a);
11     printf("%d\n", a);
12
```

(次のページに続く)

(前のページからの続き)

```
13      exit(EXIT_SUCCESS);
14  }
```

コンパイル時のエラーメッセージはそれぞれ以下のようになる。これらのメッセージから間違いの箇所を特定してみる。

★ 練習

エラーメッセージの内容を読んで間違いを見つけてみよう。

```
$ gcc -o 3-1 3-1.c
3-1.c: In function ‘main’:
3-1.c:10:5: error: expected ‘;’ before ‘exit’
     exit(EXIT_SUCCESS);
     ^~~~
```

```
$ gcc -o 3-2 3-2.c
3-2.c: In function ‘main’
3-2.c:8:5: warning: implicit declaration of function ‘qrintf’; did you mean ‘printf[-
↪Wimplicit-function-declaration]
     qrintf("Hello world.\n");
     ^~~~~~
     printf
/tmp/ccRDv9O7.o: 関数 `main' 内:
3-2.c:(.text+0x11): `qrintf' に対する定義されていない参照です
collect2: error: ld returned 1 exit status
```

```
$ gcc -o 3-3 3-3.c
3-3.c:3:10: fatal error: stdi0.h: そのようなファイルやディレクトリはありません
 #include <stdi0.h>
          ^~~~~~~~~
compilation terminated.
```

```
$ gcc -o 3-4 3-4.c
3-4.c: In function ‘main’:
3-4.c:10:10: error: ‘EXlT_SUCCE55’ undeclared (first use in this function); did you␣
↪mean ‘EXIT_SUCCESS’?
     exit(EXlT_SUCCE55);
          ^~~~~~~~~~~~
          EXIT_SUCCESS
3-4.c:10:10: note: each undeclared identifier is reported only once for each function␣
↪it appears in
```

```
$ gcc -o 3-5 3-5.c
3-5.c: In function ‘main’:
3-5.c:8:12: warning: missing terminating " character
```

(次のページに続く)

(前のページからの続き)

```
     printf("Hello world.\n);
            ^
3-5.c:8:12: error: missing terminating " character
     printf("Hello world.\n);
            ^~~~~~~~~~~~~~~~
3-5.c:10:23: error: expected ‘)’ before ‘;’ token
     exit(EXIT_SUCCESS);
                        ^
3-5.c:10:5: error: invalid use of void expression
     exit(EXIT_SUCCESS);
     ^~~~
3-5.c:11:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

```
$ gcc -o 3-6 3-6.c
3-6.c: In function ‘main’:
3-6.c:9:1: error: stray ‘\343’ in program
 ^^ef^^bf^^bd^^ef^^bf^^bd^^ef^^bf^^bd
 ^
3-6.c:9:2: error: stray ‘\200’ in program
 ^^ef^^bf^^bd^^ef^^bf^^bd^^ef^^bf^^bd
  ^
3-6.c:9:3: error: stray ‘\200’ in program
 ^^ef^^bf^^bd^^ef^^bf^^bd^^ef^^bf^^bd
   ^
```

```
$ gcc -o 3-7 3-7.c
3-7.c: In function ‘main’:
3-7.c:11:13: warning: format ‘%d’ expects argument of type ‘int *’, but argument 2␣
↪has type ‘int’ [-Wformat=]
 scanf("%d", a);
            ~^
```

3.1.2 C++ 言語　文法エラーがある場合

C++ では同じプログラムでもエラーの出方が異なる。間違いの箇所は C 言語の例と同じであるので、エラーメッセージの内容を比較してみる。

リスト 3.8　3-11.cpp

```
1 // g++ -Wall -ansi -std=c++11 -o 3-11 3-11.cpp
2 #include <cstdio>
3 #include <cstdlib>
4
```

(次のページに続く)

(前のページからの続き)

```
int main(void)
{
    printf("Hello world.\n")

    exit(EXIT_SUCCESS);
}
```

リスト 3.9　3-12.cpp

```
// g++ -Wall -ansi -std=c++11 -o 3-12 3-12.cpp
#include <cstdio>
#include <cstdlib>

int main(void)
{
    qrintf("Hello world.\n");

    exit(EXIT_SUCCESS);
}
```

リスト 3.10　3-13.cpp

```
// g++ -Wall -ansi -std=c++11 -o 3-13 3-13.cpp
#include <cstdi0>
#include <cstd1ib>

int main(void)
{
    printf("Hello world.\n");

    exit(EXIT_SUCCESS);
}
```

リスト 3.11　3-14.cpp

```
// g++ -Wall -ansi -std=c++11 -o 3-14 3-14.cpp
#include <cstdio>
#include <cstdlib>

int main(void)
{
    printf("Hello world.\n");

    exit(EXlT_SUCCE55);
}
```

リスト 3.12　3-15.cpp

```
// g++ -Wall -ansi -std=c++11 -o 3-15 3-15.cpp
#include <cstdio>
#include <cstdlib>

int main(void)
{
    printf("Hello world.\n);

    exit(EXIT_SUCCESS);
}
```

リスト 3.13　3-16.cpp

```
// g++ -Wall -ansi -std=c++11 -o 3-16 3-16.cpp
#include <cstdio>
#include <cstdlib>

int main(void)
{
    printf("Hello world.\n");  // １行コメントはこのように書く

    exit(EXIT_SUCCESS);
}
```

リスト 3.14　3-17.cpp

```
// g++ -Wall -ansi -std=c++11 -o 3-17 3-17.c
#include <cstdio>
#include <cstdlib>

int main(void)
{
    int a;

    printf("Input number -> ");
    scanf("%d", a);
    printf("%d\n", a);

    exit(EXIT_SUCCESS);
}
```

g++ によるコンパイル時のエラーメッセージはそれぞれ以下のようになる。

```
$ g++ -o 3-11 3-11.cpp
3-11.cpp: In function ‘int main()’:
3-11.cpp:10:5: error: expected ‘;’ before ‘exit’
```

(次のページに続く)

(前のページからの続き)

```
     exit(EXIT_SUCCESS);
     ^~~~
```

```
$ g++ -o 3-12 3-12.cpp
3-12.cpp: In function ‘int main()’:
3-12.cpp:8:5: error: ‘qrintf’ was not declared in this scope
     qrintf("Hello world.\n");
     ^~~~~~
3-12.cpp:8:5: note: suggested alternative: ‘printf’
     qrintf("Hello world.\n");
     ^~~~~~
     printf
```

```
$ g++ -o 3-13 3-13.cpp
3-13.cpp:3:10: fatal error: cstdi0: そのようなファイルやディレクトリはありません
 #include <cstdi0>
          ^~~~~~~~
compilation terminated.
```

```
$ g++ -o 3-14 3-14.cpp
3-14.cpp: In function ‘int main()’:
3-14.cpp:10:10: error: ‘EXlT_SUCCE55’ was not declared in this scope
     exit(EXlT_SUCCE55);
          ^~~~~~~~~~~~
3-14.cpp:10:10: note: suggested alternative: ‘EXIT_SUCCESS’
     exit(EXlT_SUCCE55);
          ^~~~~~~~~~~~
          EXIT_SUCCESS
```

```
$ g++ -o 3-15 3-15.cpp
3-15.cpp:8:12: warning: missing terminating " character
     printf("Hello world.\n);
             ^
3-15.cpp:8:12: error: missing terminating " character
     printf("Hello world.\n);
             ^~~~~~~~~~~~~~~~
3-15.cpp: In function ‘int main()’:
3-15.cpp:10:23: error: expected ‘)’ before ‘;’ token
     exit(EXIT_SUCCESS);
                         ^
```

```
$ g++ -o 3-16 3-16.cpp
3-16.cpp:9:1: error: stray ‘\343’ in program
 ^^ef^^bf^^bd^^ef^^bf^^bd^^ef^^bf^^bd
 ^
```

(次のページに続く)

(前のページからの続き)

```
3-16.cpp:9:2: error: stray ‘\200’ in program
 ^^ef^^bf^^bd^^ef^^bf^^bd^^ef^^bf^^bd
  ^
3-16.cpp:9:3: error: stray ‘\200’ in program
 ^^ef^^bf^^bd^^ef^^bf^^bd^^ef^^bf^^bd
   ^
```

```
$ g++ -o 3-17 3-17.cpp
3-17.cpp: In function ‘int main()’:
3-17.cpp:11:18: warning: format ‘%d’ expects argument of type ‘int*’, but argument 2␣
↪has type ‘int’ [-Wformat=]
     scanf("%d", a);
                  ^
```

3.2 文法エラーがない場合のデバッグ

文法エラーがない場合はコンパイル時にエラーが表示されずに終了することが多い。ただし現在のコンパイラは、文法エラーでない場合でも間違いの可能性があるときは警告を表示することがある。よって警告表示のオプションを指定しておくと警告が表示され間違いに気がつくことがある。以下のプログラムはいずれも文法的には間違いはないが正しい結果が得られず、修正すべき部分がある。

このような間違いには

- 文法の勘違いだがたまたまエラーとならない
- 単純な書き間違い
- ロジックの間違い、勘違い

などがある。

★ 練習

次に示すプログラムでそれぞれどこに問題があるか考えてみよう、そして正しく動くように直してみよう。

リスト 3.15　3-21.c

```c
// gcc -Wall -ansi -std=c99 -o 3-21 3-21.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a, b;

    printf("Input number -> ");
```

(次のページに続く)

(前のページからの続き)

```
    scanf("%d", &a);
    printf("Input number -> ");
    scanf("%d", &b);

    if (a = b) printf("a and b are equal.\n");
    else       printf("a and b are not equal.\n");

    exit(EXIT_SUCCESS);
}
```

リスト 3.16　3-22.c

```
// gcc -Wall -ansi -std=c99 -o 3-22 3-22.c
#include <stdio.h>
#include <stdlib.h>

#define         NUM         10

int main(void)
{
    int i, sum;
    int a[NUM] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (i = 1; i <= NUM; i++); {
        sum += a[i];
    }
    printf("Average is %f\n", (float)sum / NUM);

    exit(EXIT_SUCCESS);
}
```

デバッグの具体的な方法として次の 2 つがあげられる。

- プログラム内の所々に printf() を挿入して、変数の値を確認することで間違いを推定する。
- デバッグの作業をサポートする **デバッガ** という専用のプログラムがあるので、それを活用する。

プログラムの間違いの多くは、処理結果が記録される変数の値が正しい値や想定した本来なるべき値と異なることで顕在化する。よって、プログラムの様々な場所で変数の値を確認することで、どこに間違いがあるか、どのような間違いかを推測することができる。

3.3 printf() を使うデバッグ

上記のプログラムに printf() を挿入したのが次のプログラムである。挿入した行の上には目印として ////////////////////// を付加している。

★ 練習

表示される変数の値に基づいてどこが間違っているのか考えてみよう。

リスト 3.17　3-21b.c

```
// gcc -Wall -ansi -std=c99 -o 3-21b 3-21b.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a, b;

    printf("Input number -> ");
    scanf("%d", &a);
    printf("Input number -> ");
    scanf("%d", &b);

//////////////////////////////////////////
    printf("a=%d b=%d\n", a, b);

    if (a = b) printf("a and b are equal.\n");
    else       printf("a and b are not equal.\n");

//////////////////////////////////////////
    printf("a=%d b=%d\n", a, b);

    exit(EXIT_SUCCESS);
}
```

リスト 3.18　3-22b.c

```
// gcc -Wall -ansi -std=c99 -o 3-22b 3-22b.c
#include <stdio.h>
#include <stdlib.h>

#define         NUM         10

int main(void)
{
    int i, sum;
    int a[NUM] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

```

(次のページに続く)

(前のページからの続き)

```
    for (i = 1; i <= NUM; i++); {
//////////////////////////////////////////
        printf("sum=%d i=%d a[i]=%d\n", sum, i, a[i]);
        sum += a[i];
    }
//////////////////////////////////////////
    printf("sum=%d\n", sum);

    printf("Average is %f\n", (float)sum / NUM);

    exit(EXIT_SUCCESS);
}
```

printf() を使う方法は特別な知識を必要とせずすぐに実行できるが、規模の大きなプログラムになるとどこを調べるとよいのか、それを決めることも簡単ではない。例えば最近の OS のソースファイルの総行数は以下のようになる。

- 最近の Linux kernel : 2000 万行超
- Debian 5.0 : 3 億 2000 万行超
- Windows 7 : 1 億行

このような大規模なソフトウェアでは、ツールを使わずに手作業でデバッグを行うのはほぼ不可能である。規模の大きなプログラムでは次に示すデバッガを活用するほうが早く効率的に問題を解決できる。

3.4 デバッグ用のツール「デバッガ」を使うデバッグ

gcc, g++ と同じように GNU プロジェクトにより開発、公開されているデバッガとして gdb がある。ここでは gdb を使ったデバッグの基礎を取り上げる。次に示すプログラムはコンパイルは問題なく終了するが、実行するとメッセージを表示して異常終了してしまう。

リスト 3.19　3-23.c

```
// gcc -Wall -ansi -std=c99 -ggdb -o 3-23 3-23.c
#include <stdio.h>
#include <stdlib.h>

#define         NUM1        100
#define         NUM2        50

int main(void)
{
    int i, sum = 0;
    int a[NUM1+NUM2], b[NUM1+NUM2], c[NUM1+NUM2], d[NUM1+NUM2], e[NUM1+NUM2];

```

(次のページに続く)

(前のページからの続き)

```
    for (i = 0; i < NUM1+NUM2; i++) {
        a[i] = i * 10;
    }
    for (i = 0; i < NUM1+NUM2; i++) {
        b[i] = a[i] + 10;
    }
    for (i = 0; i < NUM1+NUM2; i++) {
        c[i] = b[i] / 20;
    }
    for (i = 0; i < NUM1+NUM2; i++) {
        d[i] = c[i] - 10;
    }
    for (i = 0; i < NUM1+NUM2; i++) {
        e[i] = d[i] * 10;
    }

    for (i = 0; i < NUM1*NUM2; i++) {
        sum += a[i] + b[i] + c[i] + d[i] + e[i];
    }
    printf("Sum is %d\n", sum);

    exit(EXIT_SUCCESS);
}
```

★ 練習

この後の手順を辿って実際に実行してみて、デバッガの使い方を理解しよう。

```
$ gcc -o 3-23 3-23.c
$ ./3-23
Segmentation fault (コアダンプ)
```

このプログラムを gdb を使って間違いを見つけて修正する。gdb を使うときにはデバッグ用の情報を付加した実行プログラムが必要になるため、'-ggdb' オプションを追加してコンパイルする。

```
$ gcc -ggdb -o 3-23 3-23.c
```

gdb を使うときにはコマンドライン引数として実行ファイルを指定する。メッセージが表示され起動すると、プロンプト (gdb) を表示して入力待ちとなる。

```
$ gdb ./3-23
GNU gdb (Ubuntu 8.1-0ubuntu3.1) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
```

(次のページに続く)

(前のページからの続き)

```
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./3-23...done.
(gdb)
```

r コマンド (run の意味) を指定して、デバッグ対象のプログラムを gdb の下で実行する。何も問題がなければ正常終了して入力待ちに戻る。間違いがあって異常終了したときには、どこで終了したか、終了したときの状態を表示して入力待ちに戻る。この場合は、signal SIGSEGV を受信して、Segmentation fault のエラーで異常終了したことが示されている。その下には 3-23.c 中の 31 行目に原因があることが示されている。

```
(gdb) r
Starting program: /home/any/Lecture/rp2/3-23

Program received signal SIGSEGV, Segmentation fault.
0x000055555555485b in main () at 3-23.c:31
31          sum += a[i] + b[i] + c[i] + d[i] + e[i];
```

以下では gdb の代表的なコマンドの使い方を示す。前の画面で原因の箇所は表示されているが、w コマンド (where の意味) でも原因の箇所を表示させることができる。

```
(gdb) w
#0  0x000055555555485b in main () at 3-23.c:31
```

l コマンド (list の意味) を使うと、原因の箇所周辺のソースリストを表示させることができる。

```
(gdb) l
26      for (i = 0; i < NUM1+NUM2; i++) {
27          e[i] = d[i] * 10;
28      }
29
30      for (i = 0; i < NUM1*NUM2; i++) {
31          sum += a[i] + b[i] + c[i] + d[i] + e[i];
32      }
33      printf("Sum is %d\n", sum);
34
35      exit(EXIT_SUCCESS);
```

p コマンド (print の意味) を使うと終了した時の各変数の値を表示させることができる。変数の指定はソースリスト中で書いていたそのままの表記で指定できる。gdb の実行を中断、終了するときは q コマンド (quit の意味) を指定する。

```
(gdb) p sum
$2 = 315290160

(gdb) p a[i]
$3 = 708457779

(gdb) p b[i]
$4 = 1230197573

(gdb) p i
$5 = 1392

(gdb) q
```

ここまで表示を見てみると、変数 i の値が本来の値よりも大きくなっていることに気がつくかもしれない。配列はそれぞれ a[NUM1+NUM2] のように NUM1+NUM2=150 個分しか領域を確保していないが、i = 1302 はそれを明らかに超過していることがわかる。これをもとに 31 行周辺を確認すると、30 行の for 文の終了条件が本来は NUM1+NUM2 であるべきところが、書き間違いで NUM1*NUM2 となっていて、それが原因で i の値が大きくなり、配列として確保した領域の外側をアクセスしていたことがわかる。

Segmentation fault はこの例のように、配列のアクセスに問題がある場合に表示されるので、間違いとエラーメッセージの典型的な組み合わせとして覚えておくとよい。

別の使い方として、プログラムが終了するまで一気に実行するのではなくて、少しずつ実行してその時々の変数の内容を確認できる。実行の単位は、ソースファイルを見ながら必要な点に break point という区切りを設定して、実行中に break point に達する度に中断して変数を確認できる。次に break point の使い方を見てみよう。

b コマンド (break point の意味) は、次に示すようにソースファイルの何行目に設定するかを指定する。その後、r コマンドを指定すると実行が始まり、設定した break point の位置で一旦止まる。

```
(gdb) b 3-23.c:31
Breakpoint 1 at 0x811: file 3-23.c, line 31.

(gdb) r
Starting program: /home/any/Sphinx/rp2/rp2src/3-23

Breakpoint 1, main () at 3-23.c:31
31          sum += a[i] + b[i] + c[i] + d[i] + e[i];
```

ここで変数 i の値を表示させてみると繰り返しの 1 回めであるので、初期値の 0 になっていることが確認できる。実行を再開するには c コマンド (continue の意味) を指定する。

```
(gdb) p i
$1 = 0

(gdb) c
```

(次のページに続く)

(前のページからの続き)

```
Continuing.

Breakpoint 1, main () at 3-23.c:31
31          sum += a[i] + b[i] + c[i] + d[i] + e[i];
```

再び break point の位置で止まる。繰り返しの 2 回めであるので、i = 1 になっている。以後、しばらく実行して最後に i, a[i] を表示した。

```
(gdb) p i
$2 = 1

(gdb) c
Continuing.
```

中略

```
(gdb) c
Continuing.

Breakpoint 1, main () at 3-23.c:31
31          sum += a[i] + b[i] + c[i] + d[i] + e[i];

(gdb) p i
$4 = 3

(gdb) p a[i]
$5 = 30
```

このように break point の機能を使うと、間違いの有無に関わらず、実行中の変数の変化を追跡できる。この例では break point を 1 個だけ設定したが、必要であれば 2 個以上設定することができる。

gdb には、ここで示した以外にも豊富な機能がある。gdb のその他の機能はオンラインマニュアルで確認できる。

3.5 まとめ

この章の目標

- コンパイル時のエラーメッセージや警告メッセージからプログラムの間違いの箇所を見つけられる。
- よくある間違いに対応するメッセージのパターンを理解して、間違いの箇所を早く見つけられる。
- デバッグの方法を理解して実践できる。
- デバッガの使い方を理解して、プログラムの修正に活用できる。

練習問題

1. 3-1.c の間違いを説明しなさい。
2. 3-2.c の間違いを説明しなさい。
3. 3-3.c の間違いを説明しなさい。
4. 3-4.c の間違いを説明しなさい。
5. 3-5.c の間違いを説明しなさい。
6. 3-6.c の間違いを説明しなさい。
7. 3-7.c の間違いを説明しなさい。
8. 3-21.c の間違いを説明しなさい。正しく動くように直しなさい。
9. 3-22.c の間違いを説明しなさい。正しく動くように直しなさい。
10. デバッガを使って、3-21.c, 3-22.c の実行中の変数の値を調べて、その結果に基づいてプログラムの間違いを特定しなさい。
11. 3-24.c は自然対数の底 e を漸化式　1/0! + 1/1! + 1/2! + 1/3! + ... + 1/n! により求めるプログラムであるが、計算結果が無限大に発散してしまい正しく動作しない。デバッガを使って間違いを見つけて修正しなさい。

リスト 3.20　3-24.c

```
// gcc -Wall -ansi -std=c99 -o 3-24 3-24.c
#include        <stdio.h>
#include        <stdlib.h>

int         main(void)
{
    int                 i, n, m = 1;
    float         sum = 0.0;

    printf("Input n => ");
    scanf("%d", &n);

    for (i = 0; i <= n; i++) {          // i を 0 から n まで回す
        m *= i;                         // i! (i の階乗) を求める
        sum += 1 / (float)m;          // 1/i! を合計 sum に加える
    }
    printf("%f\n", sum);

    exit(EXIT_SUCCESS);
}
```

第 4 章

プログラム作成の基本スキル

この章では、C 言語、C++ に共通するプログラム作成時に実践すると有効と思われる基本スキルを取り上げる。いずれも小さなプログラムを作成するときには無視しても大して問題にはならない。規模の大きなプログラムを作るときには少しでも間違いを減らすため、また保守性を高めて後日プログラムを見たときに容易に内容が理解できて、必要に応じて修正や機能拡張ができるようにするための知恵、手法である。

4.1 プログラムを作る能力とは

プログラムを作れるようになるには、

- 使うプログラミング言語の文法や記述法、機能に関する **知識**
- 必要とする機能を実現するために、 **命令を組み合わせる能力**

が必要となり、どちらが欠けても作ることはできない。

文法の知識は勉強して理解、覚えてしまえばそれで足りるが、命令を組み合わせる能力は覚えて終わるものではないので、数多くの練習、経験を経て身に付ける必要がある。

これはスポーツのいずれかの種目を習得するのに似ている。例として、水泳のクロールで泳げるようになるには、手足の動かし方、呼吸の仕方を知識として知っているだけでは不十分で、繰り返し練習して上手に泳げるようになる必要がある。

同様にプログラミングの学習では、文法を一通り勉強して覚えるだけでは不十分で、並行してプログラムを作る練習を数多く行って、自分が作ったプログラムと正しいプログラムを比較して、違いがあれば命令の組み合わせの正しい考え方を身につけていく必要がある。

4.2 プログラムの内容を説明するドキュメントの重要性

少し複雑な処理を行うプログラムになると、理解を容易にするためのコメント等がきちんと書かれていない場合には、たとえ自分が作成したプログラムであってもすぐには内容を思い出せないことがある。作ってから 3 ヶ月後の自分は他人と同じで、なぜそのようなプログラムにしたのか理解できず、他人のプログラムを読むのと同様に最初から注意深く読まないとわからないことが実際にある。

そのような事態を防ぐためにも変数名や関数名に関するコメント、説明のドキュメントなどをプログラムの詳細を覚えているうちに早い段階で作成しておくことが必要になる。100 行を超えるような長いプログラムや、短くても処理の内容が複雑ですぐには理解できないプログラムでは、ドキュメントが特に重要になる。ここでドキュメントと書くと分量が多い詳細な文書をイメージするかも知れないが、そこまで本格的なものでなくてもプログラムの重要な箇所に若干のコメントを書き加えるだけでもその効果は大きい。

- 作った本人には当たり前のことでも、書き表しておかないと他人にはわからないことはいくらでもある。3 ヶ月後には作った本人も忘れる。
- プログラムを作った時に何を考えたか、プログラムの表面的な記述には表れない「もとになった考え、意図」を記録する。
- 他人がプログラムを見た時にわかり難いところはどこか。
- 他人がプログラムを使うときや直す時に勘違いしやすい、間違えやすいところ「罠、落とし穴」を想像して明記する。
- 関数やプログラムの全体像がわかるような一通りの処理の流れを書く。
- 1 行ずつの細かい動作ではなく、ひとまとまりの意味のある処理のブロック毎に説明を書く。
- 定数や変数は何を表すのか、なぜその値なのか、また長さや時間の単位があれば明示する。
- 曖昧な書き方をせず、明確に書く。
- 言葉や表現を考えて誤解される可能性を低くする。

次の 2 つのプログラムは同じプログラムに、異なるコメントを付した例である。

★ 練習

第三者がプログラムを理解する際に、どちらのコメントがより有用か考えてみよう。その違いは何か。

リスト 4.1　4-1a.c

```
1  // gcc -Wall -ansi -std=c99 -o 4-1a 4-1a.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define         NUM         12          // NUM を 12 とする
6
7  int main(void)
```

(次のページに続く)

(前のページからの続き)

```
{
    int i, sum; // 変数は i, sum の 2 個を使用する
    int a[NUM] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}; // NUM 個分の配列 a[]

    for (i = 0; i < NUM; i++); { // 配列の個数分繰り返す
        sum += a[i]; // sum に i 番目の a[] を足す
    }
    printf("Average is %f\n", (float)sum / NUM); // 合計値を個数で割る

    exit(EXIT_SUCCESS);
}
```

リスト 4.2　4-1b.c

```
// gcc -Wall -ansi -std=c99 -o 4-1b 4-1b.c
#include <stdio.h>
#include <stdlib.h>

#define         NUM         12 // 配列のサイズは 12, 月毎の値を扱う

// 配列の要素の平均値を求めるプログラム
int main(void)
{
    int i, sum; // sum は合計値を入れる
    int a[NUM] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

    for (i = 0; i < NUM; i++); {
        sum += a[i]; // 配列の各要素の和を sum に求める
    }
    printf("Average is %f\n", (float)sum / NUM); // 平均値

    exit(EXIT_SUCCESS);
}
```

4.3 関数名、変数名には意味を持たせる

規模の大きなプログラムを作るときには、変数名や付随するコメントを見たときに、すぐにどういう働きをする変数や関数か理解できるような名前やコメントを付ける。特にプログラムの実行中ずっと存在する変数、プログラムの動作を決める変数などはわかりやすい名前、コメントを付ける。一方で、使い捨ての変数、局所的に短期間しか存在しない変数は簡単な名前でも問題ないことが多い。

関数名、変数名の決め方では、以下に示す方法が使われることがある。すべてを使う必要はないが、自分が良いと考えるものを取り入れるとよい。

- 重要な変数や関数には長い詳しい名前をつける。

- 一時的にしか使わない、存在しない、重要でない変数や関数には短い名前をつける。
- 変数名の最初は名詞、関数名の最初は動詞にする。
- 内容がわかるように複数の言葉を組み合わせる。
- 複数の単語の間をアンダーバー (_) で繋ぐ。
- 複数の単語を並べる際に、単語の最初の文字を大文字、残りを小文字にする。
- 変数名の最初に属性が分かる文字を付加する。例として、グローバル変数には 'g', クラスのメンバ変数 (C++ の機能) には 'c', ポインタには 'p', など。
- 抽象的な意味の単語を使わず、具体的な意味の単語を使う。
- 様々な解釈が可能な単語を使わず、誤解されない明確な意味の単語を使う。

リスト 4.3　4-1c.c

```
1  // gcc -Wall -ansi -std=c99 -o 4-1c 4-1c.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int* gIDList[10];
6
7  // 変数名や関数名の付け方の例
8
9  int main(void)
10 {
11 // 重要な変数や関数には長い詳しい名前をつける。
12
13     char MemberNameList[100][64];
14
15 // 一時的にしか使わない、存在しない、重要でない変数や関数には短い名前をつける。
16
17     int i, j, tmp;
18
19 // 変数名の最初は名詞、関数名の最初は動詞にする。
20
21     int MemberIDList[10];
22     ReadMemberID();
23
24 // 内容がわかるように複数の言葉を組み合わせる。
25
26     int HowmanyAllMembers;
27
28 // 複数の単語の間をアンダーバー (_) で繋ぐ。
29
30     int number_of_all_member;
31
```

(次のページに続く)

(前のページからの続き)

```
// 複数の単語を並べる際に、単語の最初の文字を大文字、残りを小文字にする。

    int SelectedMemberIDList[10];

// 変数名の最初に属性が分かる文字を付加する。

    int* pMemberIDList;
    pMemberIDList = gIDList;

// 抽象的な意味の単語を使わず、具体的な意味の単語を使う。

    PutValue();    // どこに出力するのかわからない、曖昧
    SaveValue();   // ファイルに記録する
    PrintValue(); // 表示する

// 様々な解釈が可能な単語を使わず、誤解されない明確な意味の単語を使う。

    SetRobot();   // ロボットの設定、どう設定するのか不明
    InitRobot(); // ロボットの初期化

    exit(EXIT_SUCCESS);
}
```

4.4 グローバル変数とローカル変数

変数をグローバル変数として宣言するか、ローカル変数として宣言するか、どちらでも選択可能な場合、特別な理由がなければなるべくローカル変数として宣言する方がよい。グローバル変数にするとプログラム内のどこでも内容を変更できてしまうので、間違いのもとになり保守時の信頼性が下がる。ローカル変数にすると、その変数を参照したり変更できるのはプログラム内の特定の部分に限定されるので保守性が向上する。更にローカル変数においても参照可能なプログラムの範囲をなるべく狭く必要最低限に限定するのがよい。

次に示す 2 つのプログラム 4-1.c, 4-2.c は同じ処理内容であるが、変数を宣言する位置が異なる。

★ 練習

どちらのプログラムが適切か、考えてみよう。その違いは何か。

リスト 4.4　4-1.c

```
// gcc -Wall -ansi -std=c99 -o 4-1 4-1.c
#include <stdio.h>
#include <stdlib.h>

#define         N         100

```

(次のページに続く)

(前のページからの続き)

```
int         i, j, a[N][N];
int         sum = 0;

void        f1(void);

int main(void)
{
    f1();

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    }

    printf("sum = %d\n", sum);

    exit(EXIT_SUCCESS);
}

void        f1(void)
{
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            a[i][j] = i * j;
        }
    }
}
```

リスト 4.5　4-2.c

```
// gcc -Wall -ansi -std=c99 -o 4-2 4-2.c
#include <stdio.h>
#include <stdlib.h>

#define         N       100

void        f1(int a[][N]);

int main(void)
{
    int         a[N][N], sum = 0;

    f1(a);

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            sum += a[i][j];
```

(次のページに続く)

(前のページからの続き)

```
        }
    }

    printf("sum = %d\n", sum);

    exit(EXIT_SUCCESS);
}

void        f1(int a[][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[i][j] = i * j;
        }
    }
}
```

for (int i = 0; i < 10; i++) { ... } のような宣言の仕方は、古い C 言語の規格では認められていなかったが、C99 以降では使えるようになったので、積極的に使うとよい。

次のプログラムでは同じ名前の変数 a が複数の場所で宣言されている。

★ 練習

それぞれの printf() の場所で、どの a の値が参照されるか考えてみよう。その後、コンパイル、実行して確認しよう。

リスト 4.6　4-3.c

```
// gcc -Wall -ansi -std=c99 -o 4-3 4-3.c
#include <stdio.h>
#include <stdlib.h>

void        f1(void);
void        f2(void);
void        f3(int a);

struct {
    int a;
} b;

int         a = 1;

int main(void)
{
    printf("a0 = %d\n", b.a = 10);
    printf("a1 = %d\n", a);
    {
```

(次のページに続く)

(前のページからの続き)

```
        int         a = 2;
        printf("a2 = %d\n", a);
        {int a = 3; printf("a3 = %d\n", a);}
        f1();
        f2();
        f3(a);
    }
    printf("a4 = %d\n", a);

    exit(EXIT_SUCCESS);
}

void        f1(void)
{
    int         a = 4;

    printf("a5 = %d\n", a);
}

void        f2(void)
{
    printf("a6 = %d\n", a);
}

void        f3(int a)
{
    printf("a7 = %d\n", a);
}
```

4.5 インデント、プログラムの整形

プログラムを記述するときに適切なインデント（字下げ）を行い、かつプログラム全体で一貫した整形のルールに則って記述することは、後の保守時の可読性を確保するのに重要な意味を持つ。変数名やコメントをわかりやすく記述する、説明のドキュメントを整備するのとあわせて、プログラムの構造を理解しやすくするために一貫した整形のルールを持つことは同じ程度に重要である。

インデントと整形のルールは、ある程度標準的な暗黙のルールがあるが、必ずしもそれらに従う必要はない。多少標準と異なっても自分が決めたルールがあって、プログラム全体がそれに従っていれば良い。プログラムの部分によって、記述の仕方が変化するのは、混乱の元になるのでよくない。

以下のプログラムはわざと字下げや空行を含めないで書いたものであるがこのような短いプログラムであっても、手を抜かずにきちんと整形のルールを適用する必要がある。短いプログラムしか書いていないと一貫した記述のルールが重要であることがわからないことがあるが、プログラミングを学ぶ初期の段階から一貫した整形のルールに従う習慣を身につけるべきである。

★ 練習

このプログラムを自分のルールに従って整形してみよう。

リスト 4.7　4-5.c

```
// gcc -Wall -ansi -std=c99 -o 4-5 4-5.c
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
typedef         unsigned char         Uchar;
int         main(int argc, char *argv[])
{
Uchar         rgb[3];
FILE         *fp;
if (2 != argc) {
fprintf(stderr, "Usage: %s file\n", argv[0]);
exit(0);
}
if ((FILE *)NULL == (fp = fopen(argv[1], "w"))) {
fprintf(stderr, "Error: [%s] can not open\n", argv[1]);
exit(1);
}
fprintf(fp, "P6\n256 256\n255\n");
for (int i = 0; i < 256; i++) {
for (int j = 0; j < 256; j++) {
rgb[0] = i;
rgb[1] = j;
rgb[2] = 255 - i;
fwrite(rgb, sizeof(Uchar), 3, fp);
}
}
fclose(fp);
exit(0);
}

```

このプログラムは PPM 形式の画像ファイルを書き出す。実行時にデータをセーブするファイル名をコマンドライン引数として一つ指定する。画像ファイルのフォーマットには、.jpg, .bmp. .png など様々なものがあるが.ppm はその一つである。データの圧縮がされないので操作が単純になり、簡潔なプログラムでも扱うことができる。

★ 練習

コンパイルして実行してみよう。出力された画像ファイル (zzz.ppm) をダブルクリックして表示してみよう。

```
$ gcc -Wall -ansi -std=c99 -o 4-5 4-5.c
$ ./4-5 zzz.ppm
```

4.6 関数のプロトタイプ宣言

関数のプロトタイプ宣言は、プログラム中で関数が呼び出される前に、その関数の返り値の型と引数の型と個数を宣言する記述である。次に示す 4-10.c はプロトタイプ宣言をしないで記述したプログラムの例である。

リスト 4.8　4-10.c

```
// gcc -Wall -ansi -std=c99 -o 4-10 4-10.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    f1();

    exit(EXIT_SUCCESS);
}

void f1(void)
{
    printf("f1\n");
    f2();
}

void f2(void)
{
    printf("f2\n");
}
```

f1() と f2() は呼び出される前は関数の実体が定義されておらず、コンパイル時に警告が表示される。この場合、暗黙の型宣言がされているとみなされて処理される。ただし警告は表示されるがコンパイルは行われ、実行することはできる。この場合はたまたま実行結果に影響は現れていないが、結果がおかしくなることもある。

```
$ gcc -o 4-10 4-10.c
4-10.c: In function ‘main’:
4-10.c:8:5: warning: implicit declaration of function ‘f1’ [-Wimplicit-function-
↪declaration]
     f1();
     ^~
4-10.c: At top level:
4-10.c:13:6: warning: conflicting types for ‘f1’
 void f1(void)
      ^~
4-10.c:8:5: note: previous implicit declaration of ‘f1’ was here
     f1();
     ^~
4-10.c: In function ‘f1’:
4-10.c:16:5: warning: implicit declaration of function ‘f2’; did you mean ‘f1’? [-
↪Wimplicit-function-declaration]
```

(次のページに続く)

(前のページからの続き)

```
     f2();
     ^~
     f1
4-10.c: At top level:
4-10.c:19:6: warning: conflicting types for ‘f2’
 void f2(void)
      ^~
4-10.c:16:5: note: previous implicit declaration of ‘f2’ was here
     f2();
     ^~
```

```
$ ./4-10
f1
f2
```

次に示す 4-10b.c のように、プログラム中で関数が呼び出される前に、その実体が定義されれば問題はなく、正しいプログラムである。ただしすべての関数について呼び出される前の実体定義が必要であるので、関数が多くなるとその依存関係に基づいて各関数の事前定義をすべて満たすのは難しくなる。

リスト 4.9　4-10b.c

```
1  // gcc -Wall -ansi -std=c99 -o 4-10b 4-10b.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void f2(void)
6  {
7      printf("f2\n");
8  }
9
10 void f1(void)
11 {
12     printf("f1\n");
13     f2();
14 }
15
16 int main(void)
17 {
18     f1();
19
20     exit(EXIT_SUCCESS);
21 }
```

次に示す 4-10c.c のように、プロトタイプ宣言をプログラムの冒頭部分にまとめて記述することで、関数間の依存関係を気にする必要がなくなる。

リスト 4.10　4-10c.c

```
// gcc -Wall -ansi -std=c99 -o 4-10c 4-10c.c
#include <stdio.h>
#include <stdlib.h>

void f1(void);
void f2(void);

int main(void)
{
    f1();

    exit(EXIT_SUCCESS);
}

void f1(void)
{
    printf("f1\n");
    f2();
}

void f2(void)
{
    printf("f2\n");
}
```

次に示す 4-10d.c はプロトタイプ宣言がないが、f3() の定義と呼び出し時で引数の個数が異なるという致命的な間違いが含まれている。この場合でも警告は表示されるが、コンパイルは行われ、プログラムは実行できてしまう。警告を無視して実行すると、一見実行結果は得られているように見えるので、大きな問題となる。

リスト 4.11　4-10d.c

```
// gcc -Wall -ansi -std=c99 -o 4-10d 4-10d.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    f3(1);

    exit(EXIT_SUCCESS);
}

int f3(int a, int b)
{
    printf("f3 %d %d\n", a, b);
    return(0);
```

(次のページに続く)

(前のページからの続き)

```
}
```

★ 練習

コンパイル、実行して確認しよう。

```
$ gcc -o 4-10d 4-10d.c
4-10d.c: In function ‘main’:
4-10d.c:8:5: warning: implicit declaration of function ‘f3’ [-Wimplicit-function-
↪declaration]
     f3(1);
     ^~
```

```
$ ./4-10d
f3 1 923427000
```

結果を見ると、1 個めの引数の値は正しく表示されるが、2 個めの引数は渡していないのでまったく意味のない値が表示されてしまう。

4-10d.c にプロトタイプ宣言を加えたのが 4-10e.c であるが、この場合は引数の個数が異なる間違いがコンパイル時に検出されエラーとなるので、容易に修正できる。

リスト 4.12　4-10e.c

```
// gcc -Wall -ansi -std=c99 -o 4-10e 4-10e.c
#include <stdio.h>
#include <stdlib.h>

int f3(int a, int b);

int main(void)
{
    f3(1);

    exit(EXIT_SUCCESS);
}

int f3(int a, int b)
{
    printf("f3 %d %d\n", a, b);
    return(0);
}
```

```
$ gcc -o 4-10e 4-10e.c
4-10e.c: In function ‘main’:
4-10e.c:10:5: error: too few arguments to function ‘f3’
     f3(1);
```

(次のページに続く)

(前のページからの続き)

```
     ^~
4-10e.c:6:5: note: declared here
 int f3(int a, int b);
     ^~
```

4.7 個々の関数に割り当てる機能の考え方

プログラムの機能を関数として実現する場合、何を関数で行って、何を行わないか、作成する前に考える必要がある。その時には次の点を考慮するとよい。

- 一つの関数では一つの機能を実現する。様々な機能を詰め込まず、シンプルな機能とする。
- 複雑な機能の関数が必要な場合は、単純な機能の関数を組み合わせる。
- 様々な状況で使えるようになるべく汎用性を持たせることを考える。
- 関数の外部に実行時に必要となるデータや定義、定数が無い方がよい。

上記の事柄を実際のプログラムで確認してみよう。次のプログラムはいずれも整数配列の要素の総和を求める sum() の様々な実装である。4-11.c ではグローバル変数の配列に対して N 個の要素の和を求める。この実装では汎用性がなく、他の場面では全く使うことができない。

リスト 4.13　4-11.c

```
1  // gcc -Wall -ansi -std=c99 -o 4-11 4-11.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define  N       10
6
7  int sum(void);
8
9  int ary[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
10
11 int main(void)
12 {
13     int result;
14
15     result = sum();
16     printf("result : %d\n", result);
17
18     exit(EXIT_SUCCESS);
19 }
20
21 int sum(void)
22 {
```

(次のページに続く)

(前のページからの続き)

```
    int sum = 0;

    for (int i = 0; i < N; i++) sum += ary[i];

    return(sum);
}
```

グローバル変数を対象とする場合でも、次の 4-11b.c のように関数の入出力の仕様を変更して引数として渡すようにすると、他の配列の和も求められるようになり、汎用性が向上する。グローバル変数であれば、引数としなくてもアクセスできるが、そこを敢えて引数にすることで汎用性が向上する。

リスト 4.14　4-11b.c

```
// gcc -Wall -ansi -std=c99 -o 4-11b 4-11b.c
#include <stdio.h>
#include <stdlib.h>

#define  N       10

int sum(int ary[]);

int ary[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int main(void)
{
    int result;

    result = sum(ary);
    printf("result : %d\n", result);

    exit(EXIT_SUCCESS);
}

int sum(int ary[])
{
    int sum = 0;

    for (int i = 0; i < N; i++) sum += ary[i];

    return(sum);
}
```

次の 4-12.c では、総和を求めた後で関数内で結果を表示しているが、これは「計算」と「表示」を一つの関数で行っていて良い実装ではない。この実装では計算はさせたいが結果の表示が不要な場合に使うことができず、汎用性がない。

リスト 4.15　4-12.c

```c
// gcc -Wall -ansi -std=c99 -o 4-12 4-12.c
#include <stdio.h>
#include <stdlib.h>

#define  N       10

int sum(int ary[]);

int main(void)
{
    int result;
    int ary[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ary2[N] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

    result = sum(ary);
    result = sum(ary2);

    exit(EXIT_SUCCESS);
}

int sum(int ary[])
{
    int sum = 0;

    for (int i = 0; i < N; i++) sum += ary[i];
    printf("result : %d\n", sum);

    return(sum);
}
```

同様の実装でも、計算と表示を分離すると次の 4-13.c となり、後で結果を表示したい場合に使うことができ汎用性が増している。残っている問題点は、計算対象の要素の個数が外部で N として定義されているので、他の個数の場合は使うことができずその点の汎用性がない。

リスト 4.16　4-13.c

```c
// gcc -Wall -ansi -std=c99 -o 4-13 4-13.c
#include <stdio.h>
#include <stdlib.h>

#define  N       10

int sum(int ary[]);

int main(void)
{
```

(次のページに続く)

(前のページからの続き)

```
    int result, result2;
    int ary[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ary2[N] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

    result = sum(ary);
    result2 = sum(ary2);
    printf("result+result2 : %d\n", result+result2);

    exit(EXIT_SUCCESS);
}

int sum(int ary[])
{
    int sum = 0;

    for (int i = 0; i < N; i++) sum += ary[i];

    return(sum);
}
```

計算対象の個数が固定されている問題は、次の 4-14.c のように関数の引数として渡すことで他の個数でも計算できるようになり汎用性が増す。この実装では関数の外部に実行時に依存するデータや定義が一切ないので様々な場合で活用することができる。

リスト 4.17　4-14.c

```
// gcc -Wall -ansi -std=c99 -o 4-14 4-14.c
#include <stdio.h>
#include <stdlib.h>

#define  N       10

int sum(int ary[], int n);

int main(void)
{
    int result, result2;
    int ary[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ary2[N+1] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 10};

    result = sum(ary, N);
    result2 = sum(ary2, N+1);
    printf("result+result2 : %d\n", result+result2);

    exit(EXIT_SUCCESS);
}

int sum(int ary[], int n)
```

(次のページに続く)

(前のページからの続き)

```
23 {
24     int sum = 0;
25
26     for (int i = 0; i < n; i++) sum += ary[i];
27
28     return(sum);
29 }
```

4.8 まとめ

この章の目標

- プログラム作成時に必要とされる様々な基本スキルを理解して、実践できるようになる
- プログラムの内容を説明するドキュメントの重要性を理解する
- 変数名には意味を持たせる必要性を理解する
- グローバル変数とローカル変数の違いを理解して適切に使い分ける
- インデント、プログラムの整形の重要性を理解して実践する
- 関数のプロトタイプ宣言の重要性を理解して実践する
- 個々の関数に割り当てる機能の考え方を理解して、応用できる

練習問題

1. 有益でないコメントやドキュメントの特徴を説明しなさい。
2. ER 学科の各学年に所属する学生の名簿を配列で作成する場合を想定して、よい変数名、悪い変数名を考えなさい。
3. 4-1.c, 4-2.c ではどちらのプログラムが適切か理由とともに説明しなさい。
4. 4-5.c を適切に整形して示しなさい。
5. 以下に示す機能を持つプログラムを単一ファイルで作成しなさい。個々の機能は必要に応じて関数として実装しなさい。(e4-1.c)

- 定数 N を 10 と定義する
- N 個の整数が記録できる配列を確保する
- キーボードから N 個の数値を入力して配列に記録する
- 最大値を求める

- 最小値を求める
- 平均値を求める
- N 個の数値の総和を求める
- N 個の数値を小さい順に並べ替える
- N 個の数値を大きい順に並べ替える
- N 個の数値の中央値 (メディアン) を求める、N が偶数の場合は中央 2 値の平均値とする
- それぞれの結果を表示する

6. 前のプログラムを分割コンパイルで生成できるように、ファイルを適切に分割しなさい。(e4-2.h, e4-2a.c, e4-2b.c, ...)
7. 前のプログラムを make で生成できるように定義ファイルを記述して make で生成しなさい。(Makefile)

第 5 章

C++ の様々な機能

ここからは C++ で C 言語から変更された機能、追加された機能を順に解説していく。同等の機能が C 言語に存在する場合は、C 言語と対比させてその違いを示す。それぞれの機能の特徴を実例を通して示すために、サンプルプログラムが数多く登場するので、その記述内容を理解するように努めよう。

5.1 入出力

最初にデータの入出力の仕方を理解しよう。5-1.c は数や文字列の入出力を行う C 言語のプログラムである。

リスト 5.1　5-1.c

```
1  // gcc -Wall -ansi -std=c99 -o 5-1 5-1.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define         N        100
6
7  int main(void)
8  {
9      int                 a, b;
10     float         f;
11     char         str[N];
12
13     printf("Input a b : ");
14     scanf("%d %d", &a, &b);
15     printf("%d %d\n", a, b);
16
17     printf("Input f : ");
18     scanf("%f", &f);
19     printf("%f\n", f);
20
21     printf("Input str : ");
22     scanf("%s", str);
```

(次のページに続く)

(前のページからの続き)

```
    printf("%s\n", str);

    exit(EXIT_SUCCESS);
}

```

実行結果は次のようになる。

```
$ ./5-2
Input a b : 3 5
3 5
Input f : 1.25
1.25
Input str : abcdefg
abcdefg
```

同じ機能を C++ で実現するのが次の 5-2.cpp である。

★練習

コンパイルして実行してみよう。 C 言語と同じ出力になることを確認しよう。

リスト 5.2　5-2.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-2 5-2.cpp
#include <iostream>
#include <cstdlib>
#include <string>

int main(void)
{
    int                 a, b;
    float         f;
    std::string         str;

    std::cout << "Input a b : ";
    std::cin >> a >> b;
    std::cout << a << ' ' << b << std::endl;

    std::cout << "Input f : ";
    std::cin >> f;
    std::cout << f << std::endl;

    std::cout << "Input str : ";
    std::cin >> str;
    std::cout << str << std::endl;

    exit(EXIT_SUCCESS);
```

(次のページに続く)

(前のページからの続き)

```
}
```

scanf() や printf() は使わず、入出力を行う **ストリームオブジェクト** として新たに用意された cin, cout を使用する。<< と >> は C 言語ではデータのシフトを行う演算子であったが、新たに入出力時にデータをストリームオブジェクトに送る機能が追加された。endl は改行を表す。文字列は C 言語では char の配列として表現されていたが、C++ では専用のデータ型 string が用意された。サイズは必要に応じて確保されるので、文字列の長さを意識して必要な大きさを確保する必要がなくなった。

5-3.c は整数や浮動小数点数を様々なフォーマットで出力する仕方を示している。

★練習

コンパイルして実行してみよう。

リスト 5.3　5-3.c

```
// gcc -Wall -ansi -std=c99 -o 5-3 5-3.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int                 a = 16;
    float         f = 1.2345;
    double         d = 1.2345;

    printf("[%d]\n", a);
    printf("[%4d]\n", a);
    printf("[%04d]\n", a);
    printf("[%o]\n", a);
    printf("[%x]\n", a);
    printf("[%04x]\n-----\n", a);

    printf("[%f]\n", f);
    printf("[%.4f]\n", f);
    printf("[%.15f]\n-----\n", f);

    printf("[%f]\n", d);
    printf("[%.4f]\n", d);
    printf("[%.15f]\n", d);
    printf("[%.30f]\n-----\n", d);

    if (16 == a) printf("equal\n");
    else         printf("not equal\n");

    if (1.2345 == f) printf("equal\n");
    else             printf("not equal\n");

```

(次のページに続く)

(前のページからの続き)

```
33      if (1.2345 == d) printf("equal\n");
34      else             printf("not equal\n");
35
36      exit(EXIT_SUCCESS);
37  }
```

%4d は 4 桁での出力、 %04d は空白を 0 で埋める指示、%o は 8 進数での出力、%x は 16 進数での出力、 %.4f は小数点以下 4 桁で出力を表している。角括弧 [,] で値がかこんであるのは、空白や空白を埋める文字がどこにあるかをわかりやすくするためである。気をつける点として、浮動小数点数は誤差を含むので比較をする時には表現精度に注意することがある。

```
$ ./5-3
[16]
[  16]
[0016]
[20]
[10]
[0010]
-----
[1.234500]
[1.2345]
[1.234500050544739]
-----
[1.234500]
[1.2345]
[1.234500000000000]
[1.23449999999999930722083263390]
-----
equal
not equal
equal
```

このプログラムの最後の比較の部分は、入出力とは別の重要な事柄を表している。整数型 int の変数が表す値は誤差が含まれないので、このプログラムのような比較は必ず正しい結果が得られる。一方、実数型 float, double の変数が表す値は値によって誤差が含まれるので、比較の結果は想定した結果になる場合とならない場合がある。今回は double 型の比較でたまたま等しいという結果になったが、これにしても厳密には同じ値を表しているわけではない (小数点以下 30 桁表示を見るとわかる)。実数型 float, double の表現ではほとんどの場合で誤差が存在するので、今回のように "==" で比較するのは危険である。常に変数が含む誤差を想定したプログラムにする必要がある。

5-3.c と同じ出力を得る C++ のプログラムが以下になる。出力のフォーマットの指定の仕方が異なるので、新たな記述の仕方を理解する必要がある。

★練習

コンパイルして実行してみよう。

リスト 5.4　5-4.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 5-4 5-4.cpp
2  #include        <iostream>
3  #include        <cstdlib>
4  #include        <iomanip>
5
6  int         main(void)
7  {
8      int                 a = 16;
9      float         f = 1.2345;
10     double         d = 1.2345;
11
12     std::ios::fmtflags flagsSaved = std::cout.flags();
13     char fillSaved = std::cout.fill();
14
15     std::cout << "[" << a << "]" << std::endl;
16     std::cout << "[" << std::setw(4) << a << "]" << std::endl;
17     std::cout.fill('0');
18     std::cout << "[" << std::setw(4) <<  a << "]" << std::endl;
19     std::cout << "[" << std::oct << a << "]" << std::endl;
20     std::cout << "[" << std::hex << a << "]" << std::endl;
21     std::cout << "[" << std::hex << std::setw(4) << a << "]" << std::endl;
22     std::cout << "-----" << std::endl;
23
24     std::cout << "[" << f << "]" << std::endl;
25     std::cout << "[" << std::fixed << std::setprecision(6) << f << "]" << std::endl;
26     std::cout << "[" << std::setprecision(15) << f << "]" << std::endl;
27     std::cout << "-----" << std::endl;
28
29     std::cout << "[" << d << "]" << std::endl;
30     std::cout << "[" << std::fixed << std::setprecision(6) << d << "]" << std::endl;
31     std::cout << "[" << std::setprecision(15) << d << "]" << std::endl;
32     std::cout << "[" << std::setprecision(30) << d << "]" << std::endl;
33
34     std::cout.flags(flagsSaved);
35     std::cout.fill(fillSaved);
36
37 // float, double の比較については同じ内容なので省略
38
39     exit(EXIT_SUCCESS);
40 }
```

11, 12 行目は出力の設定を記録する処理、26, 27 行目は記録した設定を使って当初の設定に戻す処理である。C 言語では出力の指定は 1 回限り有効であったが、C++ では一度設定するとずっと有効なものがあるので、もとの設定に戻したい場合は、このような記録と復帰の処理が必要になることがある。

```
$ ./5-4
[16]
[  16]
[0016]
[20]
[10]
[0010]
-----
[1.2345]
[1.234500]
[1.234500050544739]
-----
[1.2345000000000000]
[1.234500]
[1.2345000000000000]
[1.23449999999999993070722083263390]
```

C 言語、C++ ともに、上記以外にも多くの出力フォーマットの指定の仕方があるので、必要な場合は自分で調べるとよい。

5.2 スコープ、名前空間, using

変数の有効範囲や生存期間については 4.3.「グローバル変数とローカル変数」の節で取り上げたが、ここでは変数や関数の有効範囲について、(分割コンパイルを前提とした) 複数ファイル間での特に名前の重複の問題や扱われ方について再度確認する。C++ では名前の重複による問題を巧みに回避する仕組みが新たに取り入れられている。

5.2.1 名前の衝突、重複により生じる問題

以下の 5-7a.cpp, 5-7b.cpp では、変数 a と関数 f1() が同じ名前で定義されている。C 言語でも同じ記述が可能で、問題となる点も同じである。

★練習

コンパイルして実行してみよう。この 2 つのソースは統合されて一つの実行ファイルになるので、名前の重複によりどのような問題が生じるか見てみよう。

リスト 5.5　5-7a.cpp

```
1 // g++ -Wall -ansi -std=c++11 -o 5-7 5-7a.cpp 5-7b.cpp
2 #include <cstdio>
3 #include <cstdlib>
4
5 void        f1(void);
6 void        f2(void);
```

(次のページに続く)

(前のページからの続き)

```
int         a = 1;

int main(void)
{
    printf("main a = %d\n", a);
    f1();
    f2();

    exit(EXIT_SUCCESS);
}

void        f1(void)
{
    printf("main f1\n");
}
```

リスト 5.6　5-7b.cpp

```
// 5-7b.cpp
#include <cstdio>

static void f1(void);
// void f1(void);
// extern void f1(void);

void        f2(void);

static int a = 2;
// int a = 2;
// extern int a;

void        f1(void)
{
    printf("sub f1\n");
}

void    f2(void)
{
    printf("sub a = %d\n", a);
    f1();
}
```

5-7a.cpp では変数と関数の実体が宣言、定義されており、ここに存在する。この宣言の仕方では、他のファイルからも必要に応じて参照することができる。f2() の実体はここにはなく、他のファイルに実体があるとしてプロトタイプ宣言だけが記述されている。

```
$ ./5-7
main a = 1
main f1
sub a = 2
sub f1
```

5-7b.cpp では static 宣言が付加されているが、これにより他のファイルに存在する同じ名前の変数、関数と同時に存在させることが可能となる。static を付けた変数、関数はそのファイル内でだけ参照することができ、他のファイルからは参照できない。5-7a.cpp にある同名の変数、関数とは別の存在となる。f2() の実体はここにあり、main() からの参照を許している。

6,7 行目のコメントアウトされている宣言は外部の関数を参照する場合の記述になる。その場合、このファイルでは f1() を定義できないので、15--18 行を削除する必要がある。削除しないとエラーとなりコンパイルができない。

12, 13 行目のコメントアウトされている宣言は外部の変数を参照する場合の記述になる。12 行目は間違いでこれは通らない。変数の場合は必ず extern が必要になる。13 行目は正しい記述で 5-7a.cpp の実体を参照する。この場合値の初期化はここではできない。

このように、従来の C 言語の文法では同名の変数や関数を複数の実体として定義するには、static を付けるか、{ } で囲んだブロックの中に置く必要があり大きく制限されている。

5.2.2 名前空間の導入による名前の衝突の回避

C++ では、単一のファイルの中でも同名の変数や関数を複数の実体として定義することができる **名前空間** namespace と呼ばれる新しい概念と記法が導入されている。次に示す 5-8.cpp では名前空間を用いて、変数 a と関数 f1() を 3 個の異なる実体として定義している。

★練習

コンパイルして実行してみよう。

リスト 5.7　5-8.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 5-8 5-8.cpp
2  #include <cstdio>
3  #include <cstdlib>
4
5  int         a = 0;
6  void         f1(void);
7
8  namespace n1 {
9  int a = 1;
10 void f1(void)
11 {
12     printf("n1 a = %d\n", a);
13 }
```

(次のページに続く)

(前のページからの続き)

```
}

namespace n2 {
int a = 2;
void f1(void)
{
    printf("n2 a = %d\n", a);
}
}

int main(void)
{
    printf("a = %d\n", a);
    printf("a = %d\n", ::a);
    printf("a = %d\n", n1::a);
    printf("a = %d\n", n2::a);

    f1();
    ::f1();
    n1::f1();
    n2::f1();

    exit(EXIT_SUCCESS);
}

void        f1(void)
{
    printf(":: a = %d\n", a);
}
```

```
$ ./5-8
a = 0
a = 0
a = 1
a = 2
:: a = 0
:: a = 0
n1 a = 1
n2 a = 2
```

n1, n2 の 2 個の名前空間内とグローバルな名前空間に同じ名前で存在する。プログラムからは名前空間を指定することでそれぞれを区別して参照することができる。

名前空間は現実世界における「住所」や「所属」に相当すると考えることができる。同姓同名の人、例えば「鈴木一郎」さんが複数人いるとき、そのままでは区別ができないが、春日井市松本町の「鈴木一郎」さんと春日井市出川町の「鈴木一郎」さん、または ER の「鈴木一郎」さんと EU の「鈴木一郎」さんのように、他の何らかの属性を加えることで区別ができるようになる。同じ考え方をプログラミングの変数や関数の区別に取り入れたものとい

える。

変数と関数どちらの場合でも、何も付けない場合はグローバルな名前空間、:: を付けた場合はグローバルな名前空間、n1:: や n2:: のように名前空間を指定した場合はそこに属する実体を参照する。

5.2.3 using 宣言による名前空間の指定

よく使う名前空間は using 宣言をすることで名前空間の指定を省略することができる。5-9.cpp では using namespace n2; を記述することで、名前空間の指定をしない場合は n2 が指定されたものとして扱われる。よって、35, 36 行目の b の指定は同じ意味となる。ただし、34, 38 行目のように曖昧になる場合はエラーとなり省略できない。using 宣言は次の using 宣言が現れるまで効力を持つ。

リスト 5.8　5-9.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 5-9 5-9.cpp
2  #include <cstdio>
3  #include <cstdlib>
4
5  int         a = 0;
6  void         f1(void);
7
8  namespace n1 {
9  int a = 1;
10 void f1(void)
11 {
12     printf("n1 a = %d\n", a);
13 }
14 }
15
16 namespace n2 {
17 int a = 2;
18 void f1(void)
19 {
20     printf("n2 a = %d\n", a);
21 }
22 int b = 12;
23 void f2(void)
24 {
25     printf("n2 b = %d\n", b);
26 }
27 }
28
29 using namespace n2;
30
31 int main(void)
32 {
33 //  printf("a = %d\n", a); // この場合 ::a と n2::a の区別ができずエラー
```

(次のページに続く)

(前のページからの続き)

```
    printf("b = %d\n", b);
    printf("b = %d\n", n2::b);

//  f1(); // この場合 ::f1() と n2::f1() の区別ができずエラー
    ::f1();
    n1::f1();
    n2::f1();

    f2();
    n2::f2();

    exit(EXIT_SUCCESS);
}

void        f1(void)
{
//  printf(":: a = %d\n", a); // この場合 ::a と n2::a の区別ができずエラー
    printf(":: a = %d\n", ::a);
}
```

```
$ ./5-9
b = 12
b = 12
:: a = 0
n1 a = 1
n2 a = 2
n2 b = 12
n2 b = 12
```

5-10.cpp は 5-2.cpp に対して using namespace std; により、std:: を省略できるようにしたものである。std には入出力でよく使われる定義が多く含まれるので、このように指定することがよくある。5-10.cpp は記述量が減ってスッキリしたことがわかる。

リスト 5.9　5-10.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-10 5-10.cpp
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main(void)
{
    int                 a, b;
    float         f;
    string         str;
```

(次のページに続く)

(前のページからの続き)

```
13
14      cout << "Input a b : ";
15      cin >> a >> b;
16      cout << a << ' ' << b << endl;
17
18      cout << "Input f : ";
19      cin >> f;
20      cout << f << endl;
21
22      cout << "Input str : ";
23      cin >> str;
24      cout << str << endl;
25
26      exit(EXIT_SUCCESS);
27  }
```

```
$ ./5-10
Input a b : 3 5
3 5
Input f : 1.5
1.5
Input str : abcde
abcde
```

大規模なプログラムでは、変数や関数の個数、ファイル数が膨大になるので、名前の衝突をいかにして防ぐかは重要な課題である。名前空間はそのための有効な手法となる。例として、何らかのまとまった処理の単位ごとに異なった名前空間を割り当てるとか、作成者毎に異なった名前空間を割り当てることで名前の衝突を防ぐことができるようになる。using 宣言は不用意に使うと副作用によりトラブルの原因になるので、影響する範囲をよく理解して使う必要がある。using 宣言はヘッダで使うと、その後のすべての部分に影響を及ぼすのでヘッダでは使わない方がよいとされている。

5.3 参照、ポインタとの機能の違い

まず C 言語、C++ におけるポインタによる引数の受け渡しについて復習する。

★練習

次のプログラム中のそれぞれの printf() による出力がどうなるか考えてみよう。その後、コンパイル、実行して結果を確認しよう。

リスト 5.10　5-11.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-11 5-11.cpp
#include <cstdio>
#include <cstdlib>

void        f1(int a, int b);
void        f2(int *a, int *b);

int main(void)
{
    int a = 1, b = 2;

    f1(a, b);
    printf("a = %d, b = %d\n", a, b);
    f2(&a, &b);
    printf("a = %d, b = %d\n", a, b);

    exit(EXIT_SUCCESS);
}

void        f1(int a, int b)
{
    a = 3;
    b = 4;
    printf("a = %d, b = %d\n", a, b);
}

void        f2(int *a, int *b)
{
    *a = 5;
    *b = 6;
    printf("a = %d, b = %d\n", *a, *b);
}
```

```
$ ./5-11
a = 3, b = 4
a = 1, b = 2
a = 5, b = 6
a = 5, b = 6
```

関数 f1() の呼び出しでは、引数として main() にある a, b とは別の変数が用意され、そこに a, b の値がコピーされ f1() が呼び出される (**値渡し**)。よって、呼び出された先で a, b の値が変更されても本家には影響せず、main() の方では値が変わらない。 f2() の呼び出しでは a, b のアドレスが関数に渡され (**アドレス渡し、ポインタ**)、呼び出された先では本家と同じ場所をアクセスするので本家の値が書き換わる。

C++ では、この 2 種類の引数の渡し方に加えて、第 3 の方法として **参照** , **参照渡し** という方法が追加された。参

照渡しを用いると、ポインタの場合と同じように呼び出された先で値を変更できる。この点から、参照はポインタとほぼ同じ仕組みで、できることも同じと言える。次に示す 5-13.cpp では f3() で参照の記述が出てくるが、ポインタに比べると記述量が少なくなっていることがわかる。

f1() と f3() を比較すると、関数の引数の宣言で変数名の前に & が付加されている一点だけが異なっている。これにより f3() は参照渡しとなり、関数内で引数の値を書き換えられるようになる。

リスト 5.11　5-13.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-13 5-13.cpp
#include <cstdio>
#include <cstdlib>

void        f1(int a, int b);
void        f2(int *a, int *b); // ポインタ
void        f3(int &a, int &b); // 参照

int main(void)
{
    int a = 1, b = 2;

    f1(a, b);
    printf("a = %d, b = %d\n", a, b);

    f2(&a, &b);
    printf("a = %d, b = %d\n", a, b);

    f3(a, b);
    printf("a = %d, b = %d\n", a, b);

    exit(EXIT_SUCCESS);
}

void        f1(int a, int b) // 値渡し、新たに引数用の変数が用意される
{
    a = 3;
    b = 4;
    printf("a = %d, b = %d f1()\n", a, b);
}

void        f2(int *a, int *b) // アドレス渡し、ポインタ
{
    *a = 5;
    *b = 6;
    printf("a = %d, b = %d f2()\n", *a, *b);
}

void        f3(int &a, int &b) // 参照渡し
{
```

(次のページに続く)

(前のページからの続き)

```
    a = 7;
    b = 8;
    printf("a = %d, b = %d f3()\n", a, b);
}
```

```
$ ./5-13
a = 3, b = 4 f1()
a = 1, b = 2
a = 5, b = 6 f2()
a = 5, b = 6
a = 7, b = 8 f3()
a = 7, b = 8
```

変数を参照として宣言すると、他の変数の別名として機能するようになる。次の 5-14.cpp では b を a の参照として宣言しているが、これにより a, b のどちらを操作しても a の値を変更できる。

リスト 5.12　5-14.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-14 5-14.cpp
#include <cstdio>
#include <cstdlib>

int main(void)
{
    int a = 1, &b = a; // b は a の別名となる、実体は同じ

    a = 2; // どちらを操作しても結果は同じ
    printf("a = %d, b = %d\n", a, b);

    b = 3; // どちらを操作しても結果は同じ
    printf("a = %d, b = %d\n", a, b);

    exit(EXIT_SUCCESS);
}
```

```
$ ./5-14
a = 2, b = 2
a = 3, b = 3
```

参照はポインタと同じことができるが、機能には次に示す違いがある。

- 参照は宣言時に必ず初期化、指し示す変数を指定しなければならない。ポインタは宣言と指し示す変数の指定を別に行うことができる。
- 参照は指し示す先を後で変更することはできない。ポインタは後で指し示す先を変更できる。

次に示す 5-15.cpp ではこの機能の違いを示している。

リスト 5.13　5-15.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-15 5-15.cpp
#include <cstdio>
#include <cstdlib>

void        f1(int a , int b ); // 通常の関数呼び出し
void        f2(int *a, int *b); // アドレス渡し、ポインタ
void        f3(int &a, int &b); // 参照

int main(void)
{
    int a = 1, b = 2;

    f1(a, b);
    printf("a = %d, b = %d\n\n", a, b);

// ポインタ ////////////////////////////////////////////////

    a = 1, b = 2;
    f2(&a, &b);
    printf("a = %d, b = %d\n", a, b);

    int* c = &a; int* d = &b;
//  int *c = &a; int *d = &b; // このようにも書ける
    f2(c, d);
    printf("c = %d, d = %d\n\n", *c, *d);
    c = &b; b = 999; // ポインタが指す先は変更できる
    printf("c = %d, d = %d\n\n", *c, *d);

// 参照 ////////////////////////////////////////////////////

    a = 1, b = 2;
    f3(a, b);
    printf("a = %d, b = %d\n", a, b);

    int& e = a; int& f = b; // 初期化は必須、宣言時のみできる
//  int &e = a; int &f = b; // このようにも書ける
    f3(e, f);
    printf("e = %d, f = %d\n\n", e, f);
    e = b; b = 999; // 参照先の変更は不可、値が代入されるだけ
    printf("e = %d, f = %d\n\n", e, f);

    exit(EXIT_SUCCESS);
}

void        f1(int a, int b) // 値渡し、新たに引数用の変数が用意される
{
```

(次のページに続く)

(前のページからの続き)

```
47      a = 3;
48      b = 4;
49      printf("a = %d, b = %d f1()\n", a, b);
50  }
51
52  void        f2(int *a, int *b) // アドレス渡し、ポインタ
53  {
54      *a = 5;
55      *b = 6;
56      printf("a = %d, b = %d f2()\n", *a, *b);
57  }
58
59  void        f3(int &a, int &b) // 参照渡し
60  {
61      a = 7;
62      b = 8;
63      printf("a = %d, b = %d f3()\n", a, b);
64  }
```

```
$ ./5-15
a = 3, b = 4 f1()
a = 1, b = 2

a = 5, b = 6 f2()
a = 5, b = 6
a = 5, b = 6 f2()
c = 5, d = 6

c = 999, d = 999

a = 7, b = 8 f3()
a = 7, b = 8
a = 7, b = 8 f3()
e = 7, f = 8

e = 8, f = 999
```

ポインタを使うと内容を表す時に必ず*a のように記述する必要があり、記述が煩雑になる。一方、参照では通常の変数のように記述できるので、記述を簡潔にできるのが参照を使う一番のメリットになると言える。

参照を使う時に気をつけることとして、関数を呼び出す側では、その関数が値渡しなのか参照渡しなのか、定義を見ないと区別できないという点がある。

5.4 動的メモリ確保 new, delete、malloc, free

実行するときでないと配列の大きさが決まらない場合がある。例として、画像ファイルを表示や編集するとき、音声データを加工するとき、大量の数値に対して計算をするときなどである。そのような場合のために、C 言語、C++ ともに実行時に **動的に変数の領域を確保する**機能がある。この機能は、サイズを指定してメモリを確保する命令と、使い終わった時にその領域を解放して OS に返す命令である。それぞれ、C 言語では malloc(), free() となり、 C++ では new と delete となる。

次に示す 5-16.cpp は、C++ のプログラムであるが、メモリの確保と解放は malloc(), free() を使っている。起動後に配列の個数を入力して、指定された大きさのメモリを割り当てている。プログラムが終了する前に解放している。

★練習

コンパイルして実行してみよう。

リスト 5.14　5-16.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-16 5-16.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main(void)
{
    int *ary, n;

    cout << "Input array size => ";
    cin >> n;

    ary = (int *)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) {
        ary[i] = i;
        cout << ary[i] << " ";
    }
    cout << endl;
    free(ary);

    exit(EXIT_SUCCESS);
}
```

```
$ ./5-16
Input array size => 10
0 1 2 3 4 5 6 7 8 9
```

malloc() には必要なバイト数を指定するが、あまり大きなサイズを指定して確保できずに失敗すると、返り値は NULL になる。その場合、領域は確保されていないので、そのままアクセスしようとするとエラーになる。返り値が NULL の場合はそのまま継続して実行できないので、次の 5-17.cpp のようにエラー処理を用意しておく必要が

ある。

リスト 5.15　5-17.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-17 5-17.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main(void)
{
    int *ary, n;

    cout << "Input array size => ";
    cin >> n;

    ary = (int *)malloc(sizeof(int) * n); // C言語, int 型で n 個分の領域確保
    if (NULL == ary) { // 確保できない時のエラー処理
        cerr << "Too large size." << endl;
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < n; i++) {
        ary[i] = i;
        cout << ary[i] << " ";
    }
    cout << endl;
    free(ary); // C言語, 使い終わったら忘れずに解放する

    exit(EXIT_SUCCESS);
}
```

```
$ ./5-17
Input array size => 10
0 1 2 3 4 5 6 7 8 9
```

C++ の new と delete を使うと次の 5-18.cpp のように書ける。記述の仕方は変わっているが、やっていることは同じである。

リスト 5.16　5-18.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-18 5-18.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main(void)
{
```

(次のページに続く)

(前のページからの続き)

```
    int *ary, n;

    cout << "Input array size => ";
    cin >> n;

    ary = new int[n];
    for (int i = 0; i < n; i++) {
        ary[i] = i;
        cout << ary[i] << " ";
    }
    cout << endl;
    delete[] ary;

    exit(EXIT_SUCCESS);
}
```

```
$ ./5-18
Input array size => 10
0 1 2 3 4 5 6 7 8 9
```

こちらも確保に失敗すると返り値が NULL となるので、同じようにエラー処理が必要となる。new の場合は失敗すると、それを表す例外を発生させるので、このプログラムでは例外発生を抑制する new(nothrow) として、エラー処理ができるようにしてある。

リスト 5.17　5-19.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-19 5-19.cpp
#include <iostream>
#include <cstdlib>
#include <new>

using namespace std;

int main(void)
{
    int *ary, n;

    cout << "Input array size => ";
    cin >> n;

    ary = new(nothrow) int[n]; // C++, int 型で n 個分の領域確保
    if (NULL == ary) { // 確保できない時のエラー処理
        cerr << "Too large size." << endl;
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < n; i++) {
        ary[i] = i;
```

(次のページに続く)

(前のページからの続き)

```
        cout << ary[i] << " ";
    }
    cout << endl;
    delete[] ary; // C++, 使い終わったら忘れずに解放する

    exit(EXIT_SUCCESS);
}
```

```
$ ./5-19
Input array size => 10
0 1 2 3 4 5 6 7 8 9
```

ここまでは配列を確保するという前提でサンプルプログラムを作成したが単一の変数を確保することもできる。その場合は下記の★★★が付いている 2 箇所の書き方が変わる。

リスト 5.18　5-20.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-20 5-20.cpp
#include <iostream>
#include <cstdlib>
#include <new>

using namespace std;

int main(void)
{
    int *p;

    p = new(nothrow) int; // ★★★
    if (NULL == p) { // 確保できない時のエラー処理
        cerr << "Too large size." << endl;
        exit(EXIT_FAILURE);
    }
    *p = 1;
    cout << *p << endl;
    delete p; // ★★★

    exit(EXIT_SUCCESS);
}
```

```
$ ./5-20
1
```

malloc() や new による動的なメモリ領域確保に関する注意として、malloc() と free()、new と delete は必ずセットで使うことがある。malloc() と delete、new と free() のような組み合わせでは使用してはいけない。

5.5 定数の定義 const, #define

C 言語では定数を定義する場合には #define が使われることが多い。

リスト 5.19　5-21.c

```
// gcc -Wall -ansi -std=c99 -o 5-21 5-21.c
#include <stdio.h>
#include <stdlib.h>

#define         ARYSIZE         100

int main(void)
{
    int                 a[ARYSIZE];

    for (int i = 0; i < ARYSIZE; i++) {
        a[i] = i;
        printf("%d ", a[i]);
    }
    printf("\n");

    exit(EXIT_SUCCESS);
}
```

```
$ ./5-21
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31␣
↪32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59␣
↪60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87␣
↪88 89 90 91 92 93 94 95 96 97 98 99
```

C++ ではこのような場合に const を付けた変数を使う。 const は値を変更できない変数を表す。変更できないため定数として扱うことができる。const を付けた変数は後から代入ができないため、宣言時の初期化により必要な値を代入する。

リスト 5.20　5-22.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-22 5-22.cpp
#include <iostream>
#include <cstdlib>

const int ARYSIZE = 100;

using namespace std;

int main(void)
{
    int                 a[ARYSIZE];
```

(次のページに続く)

(前のページからの続き)

```
12
13      for (int i = 0; i < ARYSIZE; i++) {
14          a[i] = i;
15          cout << a[i] << " ";
16      }
17      cout << endl;
18
19      exit(EXIT_SUCCESS);
20  }
```

```
$ ./5-22
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31␣
↪32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59␣
↪60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87␣
↪88 89 90 91 92 93 94 95 96 97 98 99
```

C 言語にも const は存在するが、使用できる場面が C++ よりも限定されているため、C++ ほどは活用されてこなかった。例として、5-22.cpp と同じように配列のサイズを指定する 5-23.c は以前の C 言語の仕様ではコンパイル時にエラーとなり使えなかった。その後 C++ の機能を一部取り込んで仕様が拡張されてきたため、新しいコンパイラでは C++ と同じように使えるようになった。

リスト 5.21　5-23.c

```
1   // gcc -Wall -ansi -std=c99 -o 5-23 5-23.c
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   const int ARYSIZE = 100;
6
7   int main(void)
8   {
9       int                 a[ARYSIZE];
10
11      for (int i = 0; i < ARYSIZE; i++) {
12          a[i] = i;
13          printf("%d ", a[i]);
14      }
15      printf("\n");
16
17      exit(EXIT_SUCCESS);
18  }
```

```
$ ./5-23
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31␣
↪32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59␣
↪60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87␣
↪88 89 90 91 92 93 94 95 96 97 98 99
```

#define と const の違いは、コンパイル前に文字列の置き換えとして処理されるか、言語の一部としてコンパイラのチェックがされるかの違いとなる。文字列の置き換えでは想定できない副作用が生じて、かつコンパイラでは検出できないことがあるので、どちらでも記述可能な場合はなるべく const を使うのが良いとされている。

ポインタに対して const を指定する場合は、記述の仕方が複数あり、それぞれで表す内容と効果が異なるので気をつける必要がある。

リスト 5.22　5-24.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-24 5-24.cpp
#include <iostream>
#include <cstdlib>

int main(void)
{
    int                 a = 1, b = 2;
    const int*         p1 = &a;         // *p1, 変数の値が変更不可になる
    int* const         p2 = &a;         // p2, アドレスが変更不可になる
    const int* const p3 = &a;         // p3, *p3 の両方が変更不可
    int const  *p4 = &a;         // *p1, 変数の値が変更不可になる

    p1 = &b; // p1 が保持するアドレスを書き換える
//  *p1 = 3; // p1 が指し示す変数の値を書き換える、エラー

//  p2 = &b; // エラー
    *p2 = 3;

//  p3 = &b; // エラー
//  *p3 = 3; // エラー

    p4 = &b;
//  *p4 = 3; // エラー

    exit(EXIT_SUCCESS);
}
```

値が変更されては困る変数、変更されることを想定していない変数、変更すべきでない変数の定義や関数間の受け渡しでは、極力 const を付けることが推奨される。const を付けることで想定外の値の変更によるプログラムの誤動作を事前に防ぐことができる。

★練習

次のプログラムでは、文字列の長さを求める関数 int strlength(char str[]) を作成しているが正しく動作しない。原因は何か考えてみよう。コンパイル、実行してみよう。

リスト 5.23　5-25.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-25 5-25.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

const int STRBUF = 10;

int         strlength(char str[]);

int main(void)
{
    char        a[STRBUF] = "abcdefg";

    cout << a << " " << strlength(a) << endl;

    exit(EXIT_SUCCESS);
}

int         strlength(char str[])
{
    for (int i = 0; i < STRBUF; i++) {
        if (str[i] = 0) return(i);
    }
    return(-99);
}
```

```
$ ./5-25
abcdefg -99
```

strlength() では str[] を変更する機能はないので、引数の定義に const を加えると次の 5-26.cpp になる。これをコンパイルすると問題点がエラーとなって見つけられる。

リスト 5.24　5-26.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-26 5-26.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

const int STRBUF = 10;

int         strlength(const char str[]);

int main(void)
```

(次のページに続く)

(前のページからの続き)

```
{
    char        a[STRBUF] = "abcdefg";

    cout << a << " " << strlength(a) << endl;

    exit(EXIT_SUCCESS);
}

int         strlength(const char str[])
{
    for (int i = 0; i < STRBUF; i++) {
        if (str[i] = 0) return(i);
    }
    return(-99);
}
```

```
$ g++ -o 5-26 5-26.cpp
5-26.cpp: In function ‘int strlength(const char*)’:
5-26.cpp:24:22: error: assignment of read-only location ‘*(str + ((sizetype)i))’
         if (str[i] = 0) return(i);
                     ^
```

比較の == をタイプミスで = としていたため正しく動作しなかった。ただしこの代入の操作は文法的には問題ないためコンパイルは通っていた。const を付加することで想定外の代入を検出することができた。

変更されるべきでない変数には「すべて」 const を付けるとよい。それにより、

- 視覚的にそれが定数であることがわかる。
- 単純なミスによる誤動作を早い段階で検出、修正できる可能性が高まる。
- ミスとは別に、関数の呼び出し時の変数の扱いを限定できる。

5.6 同じ名前の関数、関数のオーバーロード

C 言語で次の 3 個の関数を作成して main() から呼び出すプログラムを作成してみよう。関数名の最初は "cube" (3 乗の意味) を付けることにする。

- int 型の引数 1 個を受け取って、3 乗した int の値を返り値とする。
- float 型の引数 1 個を受け取って、3 乗した float の値を返り値とする。
- double 型の引数 1 個を受け取って、3 乗した double の値を返り値とする。

C 言語では同じ名前の関数を 2 個以上プログラム内に作成することはできない。よってそれぞれの関数名は何か違う名前を付けて区別する必要がある。

リスト 5.25　5-27.c

```c
// gcc -Wall -ansi -std=c99 -o 5-27 5-27.c
#include <stdio.h>
#include <stdlib.h>

int         cubeint(int n);
float         cubefloat(float n);
double         cubedouble(double n);

int main(void)
{
    int                 i = 5;
    float         f = 5.0;
    double         d = 5.0;

    printf("%d\n", cubeint(i));
    printf("%f\n", cubefloat(f));
    printf("%f\n", cubedouble(d));

    exit(EXIT_SUCCESS);
}

int         cubeint(int n)
{
    printf("int\n");
    return(n * n * n);
}

float         cubefloat(float n)
{
    printf("float\n");
    return(n * n * n);
}

double         cubedouble(double n)
{
    printf("double\n");
    return(n * n * n);
}

```

```
$ ./5-27
int
125
float
125.000000
double
```

(次のページに続く)

(前のページからの続き)

```
125.000000
```

3 個の関数は引数や返り値の型は異なるが、処理の内容は同じなので、同じ名前を付けられると見た時にわかりやすい。例えばすべて cube() という名前にできれば変数の型によって名前を考えたり区別する必要がなくなる。C++ ではこれが可能で、引数の型、個数、並びの組み合わせが異なれば同じ名前にしても区別して呼び出すことができる。これを **関数のオーバーロード** と言う。C++ では次のように書ける。

★練習

コンパイル、実行してみよう。

リスト 5.26　5-28.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 5-28 5-28.cpp
2  #include <iostream>
3  #include <cstdlib>
4
5  using namespace std;
6
7  int         cube(int n);
8  float         cube(float n);
9  double         cube(double n);
10
11 int main(void)
12 {
13     int                 i = 5;
14     float         f = 5.0;
15     double         d = 5.0;
16
17     cout << cube(i) << endl;
18     cout << cube(f) << endl;
19     cout << cube(d) << endl;
20
21     exit(EXIT_SUCCESS);
22 }
23
24 int         cube(int n)
25 {
26     cout << "int" << endl;
27     return(n * n * n);
28 }
29
30 float         cube(float n)
31 {
32     cout << "float" << endl;
33     return(n * n * n);
34 }
35
```

(次のページに続く)

(前のページからの続き)

```
double          cube(double n)
{
    cout << "double" << endl;
    return(n * n * n);
}
```

```
$ ./5-28
int
125
float
125
double
125
```

引数の型、個数、並びの組み合わせが異なれば別の関数として定義可能という条件を利用すると、次のような関数も定義できる。

リスト 5.27　5-29.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-29 5-29.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

void        f1(int a, int b, int c);
// int        f1(int a, int b, int c); // 引数が同じで返り値だけ異なるのは不可
void        f1(int a, int b, float c);
void        f1(int a, float b, int c);
void        f1(int a, float b, float c);
void        f1(int a, float b, double c);

int main(void)
{
    int                 i = 5;
    float         f = 5.0;
    double         d = 5.0;

    f1(i, i, i);
    f1(i, i, f);
    f1(i, f, i);
    f1(i, f, f);
    f1(i, f, d);

    f1(1, 1, 1);
    f1(1, 1, (float)2.0);
    f1(1, (float)2.0, 1);
```

(次のページに続く)

(前のページからの続き)

```
29      f1(1, (float)2.0, (float)2.0);
30      f1(1, (float)2.0, 2.0); // 実数はそのままでは double とみなされる
31
32      exit(EXIT_SUCCESS);
33  }
34
35  void        f1(int a, int b, int c)
36  {
37      cout << "int int int" << endl;
38  }
39
40  void        f1(int a, int b, float c)
41  {
42      cout << "int int float" << endl;
43  }
44
45  void        f1(int a, float b, int c)
46  {
47      cout << "int float int" << endl;
48  }
49
50  void        f1(int a, float b, float c)
51  {
52      cout << "int float float" << endl;
53  }
54
55  void        f1(int a, float b, double c)
56  {
57      cout << "int float double" << endl;
58  }
```

```
$ ./5-29
int int int
int int float
int float int
int float float
int float double
int int int
int int float
int float int
int float float
int float double
```

2 番目のコメントにしてある関数はエラーとなる。引数が同じで返り値だけ異なるのは、関数呼び出し時にどれを選んでよいか決められないためである。

5.7 関数テンプレート

前節では、C++ では引数が異なれば同じ名前の関数を 2 個以上定義できることを示した。ここでは異なる引数で処理の内容が同じ場合に、1 回の記述でまとめて関数を定義することができる **関数テンプレート** の機能を説明する。

5-28.cpp では、異なる型の引数に対して同じ処理をする関数が同じ名前で定義できることを示した。ただし、その場合は関数の実体を定義したい個数分だけ繰り返して記述する必要がある。関数テンプレートを利用すると、この記述を 1 回で済ませることができる。5-28.cpp を関数テンプレートを利用して書き直すと次の 5-30.cpp のように書ける。

リスト 5.28　5-30.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-30 5-30.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

template <typename TYPE>
TYPE        cube(TYPE n);

int main(void)
{
    int                 i = 5;
    float         f = 5.0;
    double         d = 5.0;

    cout << cube(i) << endl;
    cout << cube(f) << endl;
    cout << cube(d) << endl;

    exit(EXIT_SUCCESS);
}

template <typename TYPE>
TYPE        cube(TYPE n)
{
    return(n * n * n);
}
```

```
$ ./5-30
125
125
125
```

上記の TYPE は、 **テンプレート引数** , **テンプレート仮引数** と呼ばれる。変数の型を表す仮の名前で、実際に呼び出される時に渡される引数の型で置き換えられる。例えば、int の引数で呼び出すと、その時点で TYPE を int にした関数の実体が生成され実行される。関数の実体は必要になったら生成されるので、float で呼び出されることが

なければ float を処理する関数の実体は生成されない。名前として使われている TYPE は固定されておらず、任意の文字列にすることができる。

テンプレート引数の型は、1 種類である必要はなく複数個定義することができる。次の 5-31.cpp では、TYPE1, TYPE2 で 2 種類の型を扱うことができる。このテンプレート引数の条件は、a, b は必ず同じ型で、c は何でもよい (TYPE1 と同じでもよいし、異なってもよい) となる。

★練習

コンパイル、実行してみよう。

リスト 5.29　5-31.cpp

```
// g++ -Wall -ansi -std=c++11 -o 5-31 5-31.cpp
#include <iostream>
#include <cstdlib>

using namespace std;

template <typename TYPE1, typename TYPE2>
void        f1(TYPE1 a, TYPE1 b, TYPE2 c); // a, b は同じ型、c は何でもよい

int main(void)
{
    int                 i = 5;
    float         f = 5.0;

    f1(i, i, i); // これらの組み合わせは通る
    f1(i, i, f);
    f1(f, f, i);
    f1(f, f, f);
//  f1(i, f, i); // そのままでは a, b の型が異なるので不可
    f1<float>(i, f, i); // これで TYPE1 を float として呼び出すことが可能
    f1<int, int>(i, f, i); // これで TYPE1, TYPE2 は int として呼び出す
//  f1(f, i, f);
    f1(1  , 1  , 2.0);
    f1(2.0, 2.0, 1  );
//  f1(1  , 2.0, 1  ); // a, b の型が異なるので不可

    exit(EXIT_SUCCESS);
}

template <typename TYPE1, typename TYPE2>
void        f1(TYPE1 a, TYPE1 b, TYPE2 c)
{
    cout << "TYPE1 TYPE1 TYPE2" << endl;
}
```

```
$ ./5-31
TYPE1 TYPE1 TYPE2
TYPE1 TYPE1 TYPE2
TYPE1 TYPE1 TYPE2
TYPE1 TYPE1 TYPE2
TYPE1 TYPE1 TYPE2
TYPE1 TYPE1 TYPE2
TYPE1 TYPE1 TYPE2
TYPE1 TYPE1 TYPE2
```

20 行目の呼び出しは、そのままでは a, b の型が異なるのでエラーとなる。このような場合は、 **テンプレート実引数** という記述で型を明示することで対処することができる。21 行目の関数名の直後にある <float> がそれで、この指示により TYPE1 を float として呼び出すことになる。型を明示しないと矛盾が生じる場合にテンプレート実引数を指定する。

5.8 まとめ

この章の目標

- C++ の新しい入出力の記述法がわかる。
- 名前空間の概念を理解して使い方がわかる。
- 参照とポインタの機能の違いが理解できる。
- 値渡し、アドレス渡し、参照渡しの違いがわかり適切に使い分けることができる。
- 動的なメモリ確保の記述法がわかり適切に使うことができる。
- const と #define の機能の違いが理解できる、適切に使い分けることができる。。
- 同じ名前の関数がどのような場合に定義可能か理解できる。
- 関数テンプレートの記述の仕方を理解して適切に使うことができる。

練習問題

1. cin, cout を使って整数、実数、文字列を入出力するプログラムを作成しなさい。using は使用しないで作成する。(e5-1.cpp)
2. 1. のプログラムに対して using を使って std:: を省略できるようにしなさい。(e5-2.cpp)
3. 2. のプログラム全体が名前空間 abc に属するように定義しなさい。 → 問題を次のように変更する。2. のプログラムの実行する内容を関数 void f(void) として名前空間 abc に属するように定義しなさい。main() から呼び出しなさい。(e5-3.cpp)
4. 3. のプログラムに対して using を使って abc:: を省略できるようにしなさい。(e5-4.cpp)

5. アドレス渡しで 2 個の引数の値を交換する関数 void swap1(int ＊ a, int ＊ b) を作成しなさい。main() から呼び出して動作を確認しなさい。(e5-5.cpp)

6. 参照渡しで 2 個の引数の値を交換する関数 void swap2(int &a, int &b) を作成しなさい。main() から呼び出して動作を確認しなさい。(e5-6.cpp)

7. new で確保できる int の配列の上限サイズのおおよその値を実験的に求めなさい。(e5-7.cpp)

8. new で適当な同じ大きさ n の int の配列を 2 個確保して、それぞれを n 次元のベクトルとみなして、内積を計算、表示するプログラムを作成しなさい。(e5-8.cpp)

9. char src[] の文字列を逆順に並べ替えて char dst[] にコピーする関数 void reverse(char src[], char dst[]) を作成しなさい。const の指定を適切に付加しなさい。main() から呼び出して動作を確認しなさい。実行例: "abc12345678" -> "87654321cba" (e5-9.cpp)

10. int, float, double の各型で、 2 個の引数の値を交換する関数 void myswap(int &a, int &b) 等を作成しなさい。main() から呼び出して動作を確認しなさい。(e5-10.cpp)

11. 関数テンプレートの機能を利用して、2 個の引数の値を交換する関数 void myswap(TYPE &a, TYPE &b) を作成しなさい。main() から呼び出して動作を確認しなさい。(e5-11.cpp)

第 6 章

オブジェクト指向 1、class

6.1 オブジェクト指向とは

オブジェクト指向 (Object Oriented) とは、操作や処理の対象を **モノ** として表現し、そのモノに対して指示を送ることで操作や処理を行わせる、という考え方である。オブジェクト指向はこれまでに、以下に示す様々な分野で適用されている。

- オブジェクト指向プログラミング
- オブジェクトデータベース、関係データベース
- オブジェクト指向分析設計
- オブジェクト指向モデリング
- オブジェクト指向オペレーティングシステム
- オブジェクト指向ソフトウェア工学
- オブジェクト指向ユーザーインターフェース
- オブジェクト指向プロジェクト管理

ここではプログラミング分野における適用に限定する。特定のプログラミング言語に依存しない一般的なプログラムの構成要素は、 **データ** と **コード** となる。C 言語における呼び方では変数と関数となる。

- データ（C 言語では変数、オブジェクト指向では **プロパティ** と呼ばれる）
- コード（C 言語では関数、オブジェクト指向では **メソッド** と呼ばれる）

データは処理対象、コードは処理の内容を表すと言える。

C 言語は「手続き型プログラミング言語」に分類される。変数 (データ) と関数 (コード) は別のものとして表現し、言語仕様においては両者に直接の関連はない。C 言語で記述されたプログラムの動作は、必要な計算、変換、処理を行う関数を作成しておいて、そこに処理対象の変数を送って処理をさせる、と一般化できる。

C++ は「オブジェクト指向プログラミング言語」に分類される。変数 (データ) と関数 (コード) を、**クラス class** と呼ばれる表現の仕方で、一体化して記述する。この class がオブジェクト指向を言語仕様として実現する。C++ で記述されたプログラムの動作は、変数と関数を class の枠組みで一体化して作成したオブジェクトがあって、そのオブジェクトに処理の内容を表すメッセージを送って処理をさせる、と一般化できる。

オブジェクト指向の世界では、特有の概念を表すいくつかの用語があるので以下に示す。

クラス　オブジェクト指向の考え方で表現するモノの雛形、設計図、定義。クラスの段階では実体はまだない、よって操作できない。int, float などの変数の型や構造体の定義に相当する。

インスタンス (instance)　クラスに基づいて作成した変数の実体。インスタンスを作成することで操作ができるようになる。インスタンスを作成することを **インスタンス化**と呼ぶ。int a; の a は作成されたインスタンスと言える。

オブジェクト　処理対象のモノ。実体があるインスタンスはオブジェクトである。クラスは実体がないのでオブジェクトとしないことが多い。雛形や設計図というモノとされることもある。

オブジェクト指向プログラミングでは、次に示す原理、機能が重要とされている。

カプセル化 (encapsulation)　クラスの外部で必要とされない変数や関数は公開しないで、クラス内部で閉じておくこと。カプセル化を進めることで、壊れにくい、安全なプログラムを作成できる。

インヘリタンス (inheritance)、継承　複数のクラスが階層的に定義され、上位概念を表すクラスで定義される内容を、下位クラスが受け継ぐこと。例として、動物クラスが「生きている」、「動く」、「ものを食べる」という性質を持ち、哺乳類クラスが「足がある」、「子供を生む」という性質を持つとする。それらの下位クラスとして人間クラスを定義すると、前述の 2 クラスの性質は人間クラスで改めて定義しなくても、継承の機能で自動的に定義される。

ポリモーフィズム (polymorphism)、多態性　同じ名前、見た目であっても、処理対象の変数が変わるとその処理の内容が変化すること。例として、前章のサンプルプログラム 5-27.c, 5-28.cpp では、引数の 3 乗を返す関数 cube() を作成した。C 言語では多態性を備えていないため、引数の型が変わると関数名も変える必要があった。C++ は多態性を備えるため、同一の関数名 cube() で異なる型の引数を扱うことができる。

6.2 C 言語におけるデータとコードの表現

C 言語で変数、関数がどのように記述されるかを示す例として、人間を表す構造体 human を定義して、メンバ変数への代入と表示を関数で実装してみる。class は構造体と近い性質があるので、データの例として構造体を取り上げる。

リスト 6.1　6-1.cpp

```
1 // gcc -Wall -ansi -std=c99 -o 6-1 6-1.c
2 #include         <stdio.h>
3 #include         <stdlib.h>
4 #include         <string.h>
```

(次のページに続く)

(前のページからの続き)

```
5
6   #define                 ARYSIZE                 64
7
8   #define                 MALE                0
9   #define                 FEMALE                1
10
11  #define                 JAPAN               0
12  #define                 US              1
13  #define                 UK              2
14  #define                 FRANCE                  3
15  #define                 ITALY               4
16
17  struct human {
18      char          name[ARYSIZE];
19      int          nation; // nationality 国籍
20      int                 sex;    // MALE or FEMALE
21      int                 age;
22      float         height; // meter
23      float         weight; // kilo gram
24  };
25
26  typedef struct human        Human;
27
28  void human_set(Human *p, char name[], int n, int s, int a, float h, float w);
29  void human_print(Human a); // 変数と関数に言語仕様上の関係は存在しない
30
31  int main(void)
32  {
33      struct human        a; // この行、次の行、どちらの書き方でもよい
34      Human                b;
35
36      human_print(a); // 自動で初期化はされない
37      human_print(b);
38
39      strcpy(a.name, "Chubu Taro"); // 構造体はメンバ変数に自由にアクセス可能
40      a.nation = JAPAN;
41      a.sex = MALE;
42      a.age = 20;
43      a.height = 1.75;
44      a.weight = 62.0;
45      human_print(a);
46
47      human_set(&b, "Chubu Hanako", JAPAN, FEMALE, 19, 1.65, 52.0);
48      human_print(b);
49
50      exit(EXIT_SUCCESS);
51  }
52
```

(次のページに続く)

(前のページからの続き)

```
53 void human_set(Human *p, char name[], int n, int s, int a, float h, float w)
54 {
55     strcpy(p -> name, name);
56     p -> nation = n;
57     p -> sex = s;
58     p -> age = a;
59     p -> height = h;
60     p -> weight = w;
61 }
62
63 void human_print(Human a)
64 {
65     printf("name:%s  nation:%d  sex:%d  age:%d  height:%.2f  weight:%.2f\n",
66            a.name, a.nation, a.sex, a.age, a.height, a.weight);
67 }
```

```
$ gcc -o 6-1 6-1.c
$ ./6-1
name:^^ef^^bf^^bd^^ef^^bf^^bdt3^^ef^^bf^^bd nation:0 sex:0 age:0 height:0.00 weight:0.
↪00
name:        nation:-457586160 sex:21851 age:-457587248 height:0.00 weight:0.00
name:Chubu Taro nation:0 sex:0 age:20 height:1.75 weight:60.00
name:Chubu Hanako nation:0 sex:1 age:19 height:1.65 weight:50.00
```

- 構造体のメンバ変数はプログラム内のどこでも参照、代入できる。
- 構造体 human と、関数 human_set(), human_print() は関係ない。関数名に human を付けて関連性がわかるようにしてあるが、そうしておかないと何をする関数かわからない。
- 関数 human_set(), human_print() には、引数として変数を渡すことで、それを処理させることができる。

6.3 C++ におけるデータとコードの表現

前節で示した 6-1.c と同様の機能を C++ で class を利用して作成すると次に示す 6-2.cpp になる。

リスト 6.2　6-2.cpp

```
1 // g++ -Wall -ansi -std=c++11 -o 6-2 6-2.cpp
2 #include         <iostream>
3 #include         <cstdlib>
4
5 const int        ARYSIZE = 64;
6
7 const int        MALE    = 0;
8 const int        FEMALE  = 1;
9
```

(次のページに続く)

(前のページからの続き)

```
const int         JAPAN    = 0;
const int         US       = 1;
const int         UK       = 2;
const int         FRANCE   = 3;
const int         ITALY    = 4;

using namespace std;

class human {
private:
public:
    string          name;
    int           nation; // nationality 国籍
    int                   sex;    // MALE or FEMALE
    int                   age;
    float         height; // meter
    float         weight; // kilo gram
public:
    void set(string name, int n, int s, int a, float h, float w);
    void print(); // class のメンバ関数としてデータと関連付けができる
};

int main(void)
{
    human         a, b; // class の場合はこのように書ける

    a.print(); // 自動で初期化はされない
    b.print();
/*
// class では公開されているメンバ変数しかアクセスできない
// class 外からは書き換えることはできない
    a.name = "Chubu Taro";
    a.nation = JAPAN;
    a.sex = MALE;
    a.age = 20;
    a.height = 1.75;
    a.weight = 60.0;
*/
    a.set("Chubu Taro", JAPAN, MALE, 20, 1.75, 62.0);
    a.print();

    b.set("Chubu Hanako", JAPAN, FEMALE, 19, 1.65, 52.0);
    b.print();

    exit(EXIT_SUCCESS);
}

void human::set(string str, int n, int s, int a, float h, float w)
```

(次のページに続く)

(前のページからの続き)

```
58 {
59     name = str;
60     nation = n;
61     sex = s;
62     age = a;
63     height = h;
64     weight = w;
65 }
66
67 void human::print()
68 {
69     cout << "name:" << name
70          << "  nation:" << nation
71          << "  sex:" << sex
72          << "  age:" << age
73          << "  height:" << height
74          << "  weight:" << weight << endl;
75 }
```

```
$ g++ -o 6-2 6-2.cpp
$ ./6-2
name:  nation:1  sex:0  age:-450580092  height:3.09519e-41  weight:0
name:  nation:-1893738080  sex:32744  age:0  height:0  weight:-4.86093e+22
name:Chubu Taro  nation:0  sex:0  age:20  height:1.75  weight:60
name:Chubu Hanako  nation:0  sex:1  age:19  height:1.65  weight:50
```

- class human にはメンバ変数の他に、メンバ関数として set(), print() が定義されている。C 言語では構造体とは別に、human_set(), human_print() として定義されていたが、 class では変数と一緒に定義できる。メンバ関数はそのオブジェクトの処理を専用に行う関数である。
- メンバ関数の呼び出し、実行は、a.set() のように構造体のメンバ変数の参照と同じ形式でオブジェクトを左に指定する。これでオブジェクト a が持つ関数 set() が呼び出される。これはオブジェクト a に対して、set() という処理を行うように指示していると考えられる。
- メンバ関数内でのメンバ変数の参照は変数名だけ書けばよい。これでそのオブジェクトの変数を参照、代入できる。

6.4 アクセス指定子 public, private

C++ において、カプセル化を実現するのが、class の定義中に書かれるアクセス指定子 public と private である。public を指定するとその後に記述されるメンバ変数やメンバ関数はオブジェクト外部に公開され、参照や代入が可能となる。private は非公開を指示してオブジェクト外部からは参照できなくなる。public と private ともに、次に別の指定が現れるまでずっと有効になる。また、public, private は class の定義中に何回現れてもよい。

カプセル化を確実に実現するために、メンバ変数やメンバ関数はなるべく private で指定して、public で指定するものは厳選するとよい。それにより外部でそのオブジェクトを扱うプログラムは注意すべき事柄が少なくなり間違いを減らすことができる。

6.5 メンバ関数

カプセル化を実現するために、よく使われる手法がメンバ変数は private で宣言しておいて、その変数を操作するメンバ関数を public で宣言してメンバ変数にアクセスする窓口をメンバ関数に限定するものである。次のプログラムは class human のメンバ関数で、性別を操作する関数の例である。

リスト 6.3　6-3.cpp

```
// g++ -Wall -ansi -std=c++11 -o 6-3 6-3.cpp
#include          <iostream>
#include          <cstdlib>

const int         SUCCESS = 0;
const int         ERROR   = 1;

const int         MALE    = 0;
const int         FEMALE  = 1;

// 途中略

using namespace std;

class human {
private:
    string          name;
    int           nation; // nationality 国籍
    int                   sex;    // MALE or FEMALE
    int                   age;
    float          height; // meter
    float          weight; // kilo gram
public:
    void set(string name, int n, int s, int a, float h, float w);
    void print();
    int  setsex(int s); // 性別をセットする
    int  getsex(void);  // 性別の値を読み出す
};

int main(void)
{
    human         a;

    a.setsex(MALE);
    cout << a.getsex() << endl;
```

(次のページに続く)

(前のページからの続き)

```
36
37      exit(EXIT_SUCCESS);
38  }
39
40  void human::set(string str, int n, int s, int a, float h, float w)
41  {
42  //  略
43  }
44
45  void human::print()
46  {
47  //  略
48  }
49
50  int human::setsex(int s)
51  {
52      if (MALE == s || FEMALE == s) { // 値の確認
53          sex = s;
54          return(SUCCESS);
55      }
56      else {
57          return(ERROR);
58      }
59  }
60
61  int human::getsex(void)
62  {
63      return(sex);
64  }
```

```
$ ./6-3
0
```

setsex() は性別を設定し、 getsex() は性別の設定値を読み出す。メンバ変数 sex は private で宣言されているので、sex にアクセスするには setsex(), getsex() を使うしかない。setsex() では値の確認をすることで間違った値が代入されることを防いでいる。

6.6 コンストラクタ

ここまではメンバ変数に値を代入する関数の例として、human::set() を示してきた。メンバ変数への代入を行う一般的な処理にはこの関数を使える。また変数の初期化にも使うことができる。

ただし、初期化に関しては C++ の言語仕様として class に専用の仕組みが用意されているので、それを使うのがよい。独自に作成した初期化の機能や関数は使わないことが推奨される。

用意されている専用の仕組みとは、**コンストラクタ (Constructor)** と呼ばれる初期化専用の関数で、コンストラクタに関して決められている書式に合わせて記述しておくと、オブジェクトが作成された直後に自動的に呼び出される。次のプログラムにはコンストラクタが記述されている。

リスト 6.4　6-4.cpp

```
// g++ -Wall -ansi -std=c++11 -o 6-4 6-4.cpp
#include          <iostream>
#include          <cstdlib>

const int         SUCCESS = 0;
const int         ERROR   = 1;

const int         MALE    = 0;
const int         FEMALE  = 1;

// 途中略

using namespace std;

class human {
private:
    string          name;
    int           nation; // nationality 国籍
    int                   sex;    // MALE or FEMALE
    int                   age;
    float         height; // meter
    float         weight; // kilo gram
public:
    human();
    human(string str, int n = 0, int s = 0, int a = 0,
          float h = 0, float w = 0);
    void set(string name, int n, int s, int a, float h, float w);
    void print();
    int  setsex(int s);
    int  getsex(void);
};

int main(void)
{
    human         a;
    human         b("bbb", 1, 1, 20, 160, 50);
    human         c("ccc", 0, 0);

    a.print();
    b.print();
    c.print();

    exit(EXIT_SUCCESS);
```

(次のページに続く)

(前のページからの続き)

```
44  }
45
46  human::human()
47  {
48      cout << "Constructor was called." << endl;
49      name = "noname";
50  }
51
52  human::human(string str, int n, int s, int a, float h, float w)
53  {
54      cout << "Constructor2 was called." << endl;
55      name = str;
56      nation = n;
57      sex = s;
58      age = a;
59      height = h;
60      weight = w;
61  }
62
63  void human::set(string str, int n, int s, int a, float h, float w)
64  {
65  //  略
66  }
67
68  void human::print()
69  {
70      cout << "name:" << name
71           << "  nation:" << nation
72           << "  sex:" << sex
73           << "  age:" << age
74           << "  height:" << height
75           << "  weight:" << weight << endl;
76  }
77
78  int human::setsex(int s)
79  {
80  //  略
81  }
82
83  int human::getsex(void)
84  {
85      return(sex);
86  }
```

25, 26 行目にプロトタイプ宣言が、47--62 行目に関数の実体がある。コンストラクタは、

- class 名と同じ名前の関数とする。この場合は、human::human() となる。

- 返り値はなく、void も記述しない。
- 引数はなくてもよい。指定することもできる。
- 明示的に呼び出さなくても自動的に実行される。
- コンストラクタを作成しない場合は、何もしないコンストラクタが内部で自動的に生成、実行される。

この例では、関数のオーバーロードの機能を使って、引数がある場合と、ない場合をそれぞれ記述している。また、引数がある場合については、 **デフォルト引数** という機能を使って、値を省略した場合に自動的にデフォルト値が渡されるようにしている。次に示すのは、このプログラムを実行した時の結果である。プログラムでは 36--38 行目でオブジェクトを生成して、その後 40--42 行目で設定値を表示している。結果を見ると、 print() の表示の前にコンストラクタが実行された時のメッセージが表示されている。

```
$ ./6-4
Constructor was called.
Constructor2 was called.
Constructor2 was called.
name:noname  nation:3379  sex:327  age:334  height:4.58e-41  weight:6.20e-27
name:bbb  nation:1  sex:1  age:20  height:160  weight:50
name:ccc  nation:0  sex:0  age:0  height:0  weight:0
```

次のプログラムはオブジェクトの代入が可能であることを示している。a, b を生成した後で、40 行目で b = a; として代入している。オブジェクト間の代入の操作は標準的な処理が用意されていて、メンバ変数の値が代入される。単一のメンバ変数は、この処理でほとんど問題ないが、配列やポインタでは問題が生じる。その場合は専用の代入処理を作成する必要がある (後述)。

リスト 6.5　6-5.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 6-5 6-5.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4
5  const int        SUCCESS = 0;
6  const int        ERROR   = 1;
7
8  const int        MALE    = 0;
9  const int        FEMALE  = 1;
10
11 // 途中略
12
13 using namespace std;
14
15 class human {
16 private:
17     string         name;
18     int          nation; // nationality 国籍
19     int                 sex;    // MALE or FEMALE
```

(次のページに続く)

(前のページからの続き)

```
20      int                 age;
21      float         height; // meter
22      float         weight; // kilo gram
23  public:
24      human();
25      human(string str, int n = 0, int s = 0, int a = 0,
26            float h = 0, float w = 0);
27      void set(string name, int n, int s, int a, float h, float w);
28      void print();
29      int  setsex(int s);
30      int  getsex(void);
31  };
32
33  int main(void)
34  {
35      human         a("aaa");
36      human         b;
37
38      a.print();
39      b = a;
40      b.print();
41
42      exit(EXIT_SUCCESS);
43  }
44
45  human::human()
46  {
47      cout << "Constructor was called." << endl;
48      name = "noname";
49  }
50
51  human::human(string str, int n, int s, int a, float h, float w)
52  {
53      cout << "Constructor2 was called." << endl;
54      name = str;
55      nation = n;
56      sex = s;
57      age = a;
58      height = h;
59      weight = w;
60  }
61
62  void human::set(string str, int n, int s, int a, float h, float w)
63  {
64  //  略
65  }
66
67  void human::print()
```

(次のページに続く)

(前のページからの続き)

```
{
    cout << "name:" << name
         << "  nation:" << nation
         << "  sex:" << sex
         << "  age:" << age
         << "  height:" << height
         << "  weight:" << weight << endl;
}

int human::setsex(int s)
{
//  略
}

int human::getsex(void)
{
    return(sex);
}
```

結果の出力では、a, b が同じ内容を保持していることがわかる。

```
$ ./6-5
Constructor2 was called.
Constructor was called.
name:aaa  nation:0  sex:0  age:0  height:0  weight:0
name:aaa  nation:0  sex:0  age:0  height:0  weight:0
```

動的にメモリを確保する new を使うと、個々のオブジェクトで異なるサイズの配列を持つことができる。新たなメンバ変数として、26, 27 行に配列のサイズとアドレスを保持するポインタを追加している。コンストラクタでは配列のサイズに合わせて new で領域を確保している。サイズが 0 以下の場合はアドレスを NULL にして確保しない別処理としている。print() には配列の内容を表示する処理を追加している。

リスト 6.6　6-6.cpp

```
// g++ -Wall -ansi -std=c++11 -o 6-6 6-6.cpp
#include         <iostream>
#include         <cstdlib>

const int        SUCCESS = 0;
const int        ERROR   = 1;

const int        MALE    = 0;
const int        FEMALE  = 1;

const int        ARYSIZE = 10;

// 途中略
```

(次のページに続く)

(前のページからの続き)

```
14
15  using namespace std;
16
17  class human {
18  private:
19      string          name;
20      int          nation; // nationality 国籍
21      int                 sex;    // MALE or FEMALE
22      int                 age;
23      float         height; // meter
24      float         weight; // kilo gram
25      int                 arysize;
26      int                 *ary;
27  public:
28      human();
29      human(string str, int n = 0, int s = 0, int a = 0,
30            float h = 0, float w = 0, int as = 0);
31      void set(string name, int n, int s, int a, float h, float w);
32      void print();
33      int  setsex(int s);
34      int  getsex(void);
35  };
36
37  int main(void)
38  {
39      human         a("aaa");
40      a.print();
41
42      human         b("bbb", 1, 1, 0, 0, 0, 10);
43      b.print();
44
45      human         c("ccc", 1, 1, 0, 0, 0, 20);
46      c.print();
47
48      exit(EXIT_SUCCESS);
49  }
50
51  human::human()
52  {
53      cout << "Constructor was called." << endl;
54      name = "noname";
55
56      arysize = ARYSIZE;
57      ary = new int[ARYSIZE];
58      for (int i = 0; i < arysize; i++) ary[i] = i;
59  }
60
61  human::human(string str, int n, int s, int a, float h, float w, int as)
```

(次のページに続く)

(前のページからの続き)

```
{
    cout << "Constructor2 was called." << endl;
    name = str;
    nation = n;
    sex = s;
    age = a;
    height = h;
    weight = w;
    arysize = as;
    ary = NULL;
    if (0 < arysize) {
        ary = new int[arysize];
        for (int i = 0; i < arysize; i++) ary[i] = i;
    }
    else {
        arysize = 0;
        ary = NULL;
    }
}

void human::set(string str, int n, int s, int a, float h, float w)
{
//  略
}

void human::print()
{
    cout << "name:" << name
         << "  nation:" << nation
         << "  sex:" << sex
         << "  age:" << age
         << "  height:" << height
         << "  weight:" << weight << "  address:" << ary << endl;
    if (0 < arysize) {
        for (int i = 0; i < arysize; i++) cout << ary[i] << " ";
        cout << endl;
    }
}

int human::setsex(int s)
{
//  略
}

int human::getsex(void)
{
    return(sex);
}
```

次に示す実行結果では、異なるサイズの配列の内容が表示されている。

```
$ ./6-6
Constructor2 was called.
name:aaa  nation:0  sex:0  age:0  height:0  weight:0  address:0
Constructor2 was called.
name:bbb  nation:1  sex:1  age:0  height:0  weight:0  address:0x55f0c3990280
0 1 2 3 4 5 6 7 8 9
Constructor2 was called.
name:ccc  nation:1  sex:1  age:0  height:0  weight:0  address:0x55f0c39902b0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

ここまでで、動的に領域を確保する配列を持つオブジェクトの生成と操作まで実現できた。しかし、動的に確保した領域やポインタが含まれると、まだ次のことができない。

- 確保した領域を解放する処理がない。
- 関数に渡す仮引数や、初期化時に渡す引数が正しく処理できない。
- 代入の処理が正しくできない。

6.7 デストラクタ

オブジェクト生成時の初期化の処理を行うコンストラクタと対の関係となる、オブジェクトが消滅する時の後片付けの処理を行う **デストラクタ (Destructor)** がある。コンストラクタと同じように、デストラクタはオブジェクトが消滅する時に自動的に呼び出される。

- class 名と同じ名前の関数の前に '~' を付ける。この場合は、human::~human() となる。
- 返り値はなく、void も記述しない。
- 引数はない。
- 明示的に呼び出さなくても自動的に実行される。
- デストラクタを作成しない場合は、何もしないデストラクタが内部で自動的に生成、実行される。

次のプログラムは class human にデストラクタを記述したものである。main() 中の 37, 45 行の {, } は、デストラクタのメッセージを表示するために追加している。

リスト 6.7　6-7.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 6-7 6-7.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4
5  const int        SUCCESS = 0;
6  const int        ERROR   = 1;
```

(次のページに続く)

(前のページからの続き)

```
7
8  const int         MALE    = 0;
9  const int         FEMALE  = 1;
10
11 // 途中略
12
13 using namespace std;
14
15 class human {
16 private:
17     string          name;
18     int           nation; // nationality 国籍
19     int                 sex;    // MALE or FEMALE
20     int                 age;
21     float         height; // meter
22     float         weight; // kilo gram
23 public:
24     human();
25     human(string str, int n = 0, int s = 0, int a = 0,
26           float h = 0, float w = 0);
27     ~human();
28     void set(string name, int n, int s, int a, float h, float w);
29     void print();
30     int  setsex(int s);
31     int  getsex(void);
32 };
33
34 int main(void)
35 {
36     {
37     human         a;
38     human         b("bbb", 1, 1, 20, 160, 50);
39     human         c("ccc", 0, 0);
40
41     a.print();
42     b.print();
43     c.print();
44     }
45
46     exit(EXIT_SUCCESS);
47 }
48
49 human::human()
50 {
51     cout << "Constructor was called." << endl;
52     name = "noname";
53 }
54
```

(次のページに続く)

(前のページからの続き)

```
human::human(string str, int n, int s, int a, float h, float w)
{
    cout << "Constructor2 was called." << endl;
    name = str;
    nation = n;
    sex = s;
    age = a;
    height = h;
    weight = w;
}

human::~human()
{
    cout << "Destructor was called. " << name << endl;
}

void human::set(string str, int n, int s, int a, float h, float w)
{
//  略
}

void human::print()
{
    cout << "name:" << name
         << "  nation:" << nation
         << "  sex:" << sex
         << "  age:" << age
         << "  height:" << height
         << "  weight:" << weight << endl;
}

int human::setsex(int s)
{
//  略
}

int human::getsex(void)
{
    return(sex);
}
```

実行結果が以下になる。デストラクタはオブジェクトを生成した順とは逆に実行される。すなわち後から生成されたものが先に破棄される。

```
$ ./6-7
Constructor was called.
Constructor2 was called.
```

(次のページに続く)

(前のページからの続き)

```
Constructor2 was called.
name:noname  nation:8938  sex:326  age:8879  height:4.57e-41  weight:5.69e-07
name:bbb  nation:1  sex:1  age:20  height:160  weight:50
name:ccc  nation:0  sex:0  age:0  height:0  weight:0
Destructor was called. ccc
Destructor was called. bbb
Destructor was called. noname
```

デストラクタでは、通常実行する必要がある処理はほとんどないが、唯一重要な処理として new で確保した領域の解放がある。new で確保していた領域がある場合は、必ず delete で解放しなくてはならない。これを忘れるとメモリリークとなり、使えるメモリがリークの度に徐々に少なくなる。次に示すのは new で配列を確保する 6-6.cpp に delete で配列を解放する処理を行うデストラクタを追加したものである。main() 中の 41, 48 行の {, } は、デストラクタのメッセージを表示するために追加している。

リスト 6.8　6-8.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 6-8 6-8.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4
5  const int        SUCCESS = 0;
6  const int        ERROR   = 1;
7
8  const int        MALE    = 0;
9  const int        FEMALE  = 1;
10
11 const int        ARYSIZE = 10;
12
13 // 途中略
14
15 using namespace std;
16
17 class human {
18 private:
19     string         name;
20     int          nation; // nationality 国籍
21     int                 sex;    // MALE or FEMALE
22     int                 age;
23     float        height; // meter
24     float        weight; // kilo gram
25     int                 arysize;
26     int                 *ary;
27 public:
28     human();
29     human(string str, int n = 0, int s = 0, int a = 0,
30           float h = 0, float w = 0, int as = 0);
31     ~human();
```

(次のページに続く)

(前のページからの続き)

```
    void set(string name, int n, int s, int a, float h, float w);
    void print();
    int  setsex(int s);
    int  getsex(void);
};

int main(void)
{
  {
    human        a("aaa");
    a.print();
    human        b("bbb", 1, 1, 0, 0, 0, 10);
    b.print();
    human        c("ccc", 1, 1, 0, 0, 0, 20);
    c.print();
  }
    exit(EXIT_SUCCESS);
}

human::human()
{
    cout << "Constructor was called." << endl;
    name = "noname";

    arysize = ARYSIZE;
    ary = new int[ARYSIZE];
    for (int i = 0; i < arysize; i++) ary[i] = i;
}

human::human(string str, int n, int s, int a, float h, float w, int as)
{
    cout << "Constructor2 was called." << endl;
    name = str;
    nation = n;
    sex = s;
    age = a;
    height = h;
    weight = w;
    arysize = as;
    ary = NULL;
    if (0 < arysize) {
        ary = new int[arysize];
        for (int i = 0; i < arysize; i++) ary[i] = i;
    }
    else {
        arysize = 0;
        ary = NULL;
    }
```

(次のページに続く)

(前のページからの続き)

```
}

human::~human()
{
    cout << "Destructor was called. " << name << endl;
    delete[] ary;
}

void human::set(string str, int n, int s, int a, float h, float w)
{
//  略
}

void human::print()
{
    cout << "name:" << name
         << "  nation:" << nation
         << "  sex:" << sex
         << "  age:" << age
         << "  height:" << height
         << "  weight:" << weight << "  address:" << ary << endl;
    if (0 < arysize) {
        for (int i = 0; i < arysize; i++) cout << ary[i] << " ";
        cout << endl;
    }
}

int human::setsex(int s)
{
//  略
}

int human::getsex(void)
{
    return(sex);
}
```

```
$ ./6-8
Constructor2 was called.
name:aaa  nation:0  sex:0  age:0  height:0  weight:0  address:0
Constructor2 was called.
name:bbb  nation:1  sex:1  age:0  height:0  weight:0  address:0x559e085e4280
0 1 2 3 4 5 6 7 8 9
Constructor2 was called.
name:ccc  nation:1  sex:1  age:0  height:0  weight:0  address:0x559e085e42b0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Destructor was called. ccc
```

(次のページに続く)

(前のページからの続き)

```
Destructor was called. bbb
Destructor was called. aaa
```

6.8 コピーコンストラクタ

オブジェクトの生成時にはコンストラクタが呼び出されるが、他のオブジェクトの内容をコピーして初期化する場合は、通常のコンストラクタではない別のコンストラクタが呼び出される。これを **コピーコンストラクタ (Copy Constructor)** と呼ぶ。コピーコンストラクタが呼び出されるのは、

- オブジェクト作成後の初期化時に他のオブジェクトの内容をコピーするとき
- 関数呼び出し時に値渡しで、仮引数にコピーが作成されるとき

である。

コピーコンストラクタを明示的に指定しない場合は、標準的な処理を行うコピーコンストラクタが自動的に生成され実行される。標準的な処理とは、各メンバ変数に対してコピー元の値を代入する。配列やポインタが含まれない場合はこれで問題ないが、含まれると問題が生じる。次のプログラムでその具体例を示す。これまでと同じ class human を扱うが、問題点をわかりやすくするため、関係ないメンバ変数や関数は削除してある。

リスト 6.9　6-9.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 6-9 6-9.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4
5  using namespace std;
6
7  class human {
8  private:
9      int                 arysize;
10     int                 *ary;
11 public:
12     human(int as = 0);
13     ~human();
14     void print();
15     void setary(int idx, int val);
16 };
17
18 int main(void)
19 {
20     {
21     human         a;
22     a.print();
23
```

(次のページに続く)

(前のページからの続き)

```
    human        b(20);
    b.print();

    human       c = b;
    c.print();

    cout << endl;
    b.setary(7, 99);
    b.print();
    c.print();
    }
    exit(EXIT_SUCCESS);
}

human::human(int as)
{
    cout << "Constructor was called." << endl;
    arysize = as;
    ary = NULL;
    if (0 < arysize) {
        ary = new int[arysize];
        for (int i = 0; i < arysize; i++) ary[i] = i;
    }
    else {
        arysize = 0;
        ary = NULL;
    }
}

human::~human()
{
    cout << "Destructor was called. " << endl;
    delete[] ary;
}

void human::print()
{
    cout << arysize << "  address:" << ary << endl;
    if (0 < arysize) {
        for (int i = 0; i < arysize; i++) cout << ary[i] << " ";
        cout << endl;
    }
}

void human::setary(int idx, int val)
{
    ary[idx] = val;
}
```

以下が実行結果である。c には b の内容をコピーして初期化している。b の配列の内容を一部書き換えているが、その後 b, c の配列を表示すると、c の配列の要素も同じ値に変わってしまっている。これは初期化の時に b の配列の先頭アドレスがコピーされたため、 c の配列は b と同じ場所を指しているためである。本来 c の配列は独自の場所を確保する必要があるが、それができていないことがわかる。この、内容ではなく先頭アドレスだけがコピーされるのが問題点である。標準で用意されるコピーコンストラクタではこの現象を防ぐことができないので、配列やポインタを含む場合は、それを正しく処理するコピーコンストラクタを独自に用意する必要がある。

```
$ ./6-9
Constructor was called.
0  address:0
Constructor was called.
20  address:0x56182bee7280
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20  address:0x56182bee7280
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

20  address:0x56182bee7280
0 1 2 3 4 5 6 99 8 9 10 11 12 13 14 15 16 17 18 19
20  address:0x56182bee7280
0 1 2 3 4 5 6 99 8 9 10 11 12 13 14 15 16 17 18 19
Destructor was called.
Destructor was called.
Destructor was called.
```

次に示すのは、正しくコピーの処理をするコピーコンストラクタを実装したプログラムである。

リスト 6.10　6-10.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 6-10 6-10.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4
5  using namespace std;
6
7  class human {
8  private:
9      int                 arysize;
10     int                 *ary;
11 public:
12     human(int as = 0);
13     human(const human& org);
14     ~human();
15     void print();
16     void setary(int idx, int val);
17 };
18
19 int main(void)
20 {
```

(次のページに続く)

(前のページからの続き)

```
    {
    human        a;
    a.print();

    human        b(20);
    b.print();

    human        c = b;   // b の内容をコピーして初期化する、代入とは異なる
//  human        c(b);    // こう書いても同じ意味になる
    c.print();

    human        d;
    d = b;                // これは代入
    d.print();

    cout << endl;
    b.setary(7, 99);
    b.print();
    c.print();
    d.print();            // 代入ではコピーコンストラクタは使われない、別処理
    }
    exit(EXIT_SUCCESS);
}

human::human(int as)
{
    cout << "Constructor was called." << endl;
    arysize = as;
    ary = NULL;
    if (0 < arysize) {
        ary = new int[arysize];
        for (int i = 0; i < arysize; i++) ary[i] = i;
    }
    else {
        arysize = 0;
        ary = NULL;
    }
}

human::human(const human& org)
{
    cout << "Copy constructor was called." << endl;
    arysize = org.arysize;
    ary = new int[arysize];
    for (int i = 0; i < arysize; i++) ary[i] = org.ary[i];
}

human::~human()
```

(次のページに続く)

(前のページからの続き)

```
69 {
70     cout << "Destructor was called. " << endl;
71     delete[] ary;
72 }
73
74 void human::print()
75 {
76     cout << arysize << "  address:" << ary << endl;
77     if (0 < arysize) {
78         for (int i = 0; i < arysize; i++) cout << ary[i] << " ";
79         cout << endl;
80     }
81 }
82
83 void human::setary(int idx, int val)
84 {
85     ary[idx] = val;
86 }
```

独自に用意したコピーコンストラクタでは、

- 配列のサイズを元からコピーして、
- 先頭アドレスをコピーするのではなく、new で領域を確保して、
- そこに元の配列の値をコピーしている

これでコピー元の持つ配列とは別の場所を確保して同じ値をそこに保持する。実行結果が以下である。途中、コピーコンストラクタが呼び出されていることがわかる。また、b の配列の要素を書き換えても、c の要素は影響を受けないことがわかる。ただし、コピーコンストラクタでは代入の処理はカバーしないので、代入の処理が必要な場合は、同じ要領で独自の処理を用意する必要がある。

```
$ ./6-10
Constructor was called.
0  address:0
Constructor was called.
20  address:0x56270af6e280
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Copy constructor was called.
20  address:0x56270af6e2e0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Constructor was called.
20  address:0x56270af6e280
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

20  address:0x56270af6e280
0 1 2 3 4 5 6 99 8 9 10 11 12 13 14 15 16 17 18 19
```

(次のページに続く)

(前のページからの続き)

```
20  address:0x56270af6e2e0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20  address:0x56270af6e280
0 1 2 3 4 5 6 99 8 9 10 11 12 13 14 15 16 17 18 19
Destructor was called.
Destructor was called.
Destructor was called.
Destructor was called.
```

6.9 this, メンバ関数の呼び出し元のオブジェクトを指し示すポインタ

メンバ関数を呼び出したとき、そのメンバ関数が属するオブジェクトを指し示す特殊なポインタ **this** が自動的に用意される。これはメンバ関数内で、呼び出し元のオブジェクトを参照、操作する時に使用する。

次のプログラムは、N 次元のベクタを扱うクラス myvector を実装したサンプルプログラムである。メンバ変数には次元数を表す int dim と、各次元の値を記憶する float ＊ ary がある。前節までの内容を反映して、必要な処理を行うコンストラクタ、デストラクタ、コピーコンストラクタを定義している。

リスト 6.11　6-11.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 6-11 6-11.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4
5  using namespace std;
6
7  const float V0[] = {0, 0, 0};
8
9  class myvector {
10 private:
11     int                 dim;
12     float         *ary;
13 public:
14     myvector(const int dnum = 3, const float a[] = V0);
15     myvector(const myvector& org);
16     ~myvector();
17     void print();
18     void setvec(const float a[]);
19     void getvec(float a[]);
20     void f1(int dim);
21 };
22
23 void f2(myvector *p);
24
25 int main(void)
```

(次のページに続く)

(前のページからの続き)

```
{
    myvector        v1;
    v1.print();

    myvector        v2(3);
    v2.print();

    float a[] = {1, 2, 3};
    myvector      v3(3, a);
    v3.print();

    v3.f1(10);

    exit(EXIT_SUCCESS);
}

myvector::myvector(const int dnum, const float a[])
{
    dim = dnum;
    ary = NULL;
    if (0 < dim) {
        ary = new float[dim];
        for (int i = 0; i < dim; i++) ary[i] = a[i];
    }
    else {
        dim = 0;
        ary = NULL;
    }
}

myvector::myvector(const myvector& org)
{
    dim = org.dim;
    ary = new float[dim];
    for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
}

myvector::~myvector()
{
    delete[] ary;
}

void myvector::print()
{
//  cout << "dim: " << dim << endl;          // 普通はこう書くが、
    cout << "dim: " << this -> dim << endl; // こう書くこともできる
    if (0 < dim) {
        for (int i = 0; i < dim; i++) cout << ary[i] << " ";
```

(次のページに続く)

(前のページからの続き)

```
74          cout << endl;
75      }
76      cout << "Adrs: " << this << endl; // 呼び出し元のオブジェクトのアドレス
77
78      f2(this);       // 別の関数に呼び出し元のオブジェクトを渡して処理できる
79  }
80
81  void myvector::setvec(const float a[])
82  {
83      for (int i = 0; i < dim; i++) ary[i] = a[i];
84  }
85
86  void myvector::getvec(float a[])
87  {
88      for (int i = 0; i < dim; i++) a[i] = ary[i];
89  }
90
91  void myvector::f1(int dim)
92  {
93      cout << dim << endl;  // this により引数とメンバ変数を区別できる
94      cout << this -> dim << endl;
95  }
96
97  void f2(myvector *p)
98  {
99      cout << "Adrs: " << p << endl;
100 }
```

メンバ関数 print() と f1(), f2() に this の使用例がある。

- メンバ関数内でメンバ変数を参照する場合は、変数名だけ記述すればよいが、this ポインタ経由でも同じように参照できる (71, 72 行)。
- this によりオブジェクトの先頭アドレスを読み出すことができる (77 行)。
- 他の関数にオブジェクトを引数として渡して処理をさせることができる (79 行)。
- 引数とメンバ変数の名前が重複した場合に、メンバ変数を区別して指定できる (93, 94 行)。

以下は実行結果である。

```
$ ./6-11
dim: 3
0 0 0
Adrs: 0x7fff542a3b90
Adrs: 0x7fff542a3b90
dim: 3
0 0 0
```

(次のページに続く)

(前のページからの続き)

```
Adrs: 0x7fff542a3ba0
Adrs: 0x7fff542a3ba0
dim: 3
1 2 3
Adrs: 0x7fff542a3bb0
Adrs: 0x7fff542a3bb0
10
3
```

this ポインタを使う機会はそれ程多くないが、これを使わないと実現できない機能がある。次に示す演算子のオーバーロードでは this ポインタを使った実装例が出てくる。

6.10 演算子のオーバーロード

コピーコンストラクタの節では、動的に確保する配列を持つオブジェクトにおいて、その配列を適切に扱う専用の処理を備えたコピーコンストラクタの必要性と具体的な処理の内容を取り上げた。コピーコンストラクタにより、初期化時に別のオブジェクトの内容をコピーすることはできるようになった。しかし代入の処理では、正しく動的に確保した配列をまだ扱えないことを示した。正しく扱えない理由は、標準的に用意される代入の処理は、標準的なコピーコンストラクタと同じように個々のメンバ変数の値を単純にコピー先に書き込むからである。

この節では、演算子のオーバーロードを取り上げる。C++ では、最初から用意されている様々な演算子 (=, +, -, *, / など) に対して、新しい機能を定義することができる。ここではその対象として、代入演算子 = にクラス固有の機能を追加する。動的に確保する配列を持つクラス myvector に対して、適切に代入処理が行えるようにする。

次のプログラムでは、標準的な代入の処理ではどのような問題が生じるかを確認する。

リスト 6.12　6-12.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 6-12 6-12.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4
5  using namespace std;
6
7  const float V0[] = {0, 0, 0};
8
9  class myvector {
10 private:
11     int                 dim;
12     float         *ary;
13 public:
14     myvector(const int dnum = 3, const float a[] = V0);
15     myvector(const myvector& org);
16     ~myvector();
17     void print();
```

(次のページに続く)

(前のページからの続き)

```
18      void setvec(const float a[]);
19      void getvec(float a[]);
20  };
21
22  int main(void)
23  {
24      myvector        v1, v2;
25      v2 = v1;
26      v1.print();
27      v2.print();
28
29      float a[] = {1, 2, 3};
30      v1.setvec(a);
31      v1.print();
32      v2.print();
33
34      exit(EXIT_SUCCESS);
35  }
36
37  myvector::myvector(const int dnum, const float a[])
38  {
39      dim = dnum;
40      ary = NULL;
41      if (0 < dim) {
42          ary = new float[dim];
43          for (int i = 0; i < dim; i++) ary[i] = a[i];
44      }
45      else {
46          dim = 0;
47          ary = NULL;
48      }
49  }
50
51  myvector::myvector(const myvector& org)
52  {
53      dim = org.dim;
54      ary = new float[dim];
55      for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
56  }
57
58  myvector::~myvector()
59  {
60      delete[] ary;
61  }
62
63  void myvector::print()
64  {
65      cout << "dim: " << dim << endl;
```

(次のページに続く)

(前のページからの続き)

```
66      if (0 < dim) {
67          for (int i = 0; i < dim; i++) cout << ary[i] << " ";
68          cout << endl;
69      }
70      cout << "Adrs: " << this << endl << endl;
71  }
72
73  void myvector::setvec(const float a[])
74  {
75      for (int i = 0; i < dim; i++) ary[i] = a[i];
76  }
77
78  void myvector::getvec(float a[])
79  {
80      for (int i = 0; i < dim; i++) a[i] = ary[i];
81  }
```

myvector 型のオブジェクト v1, v2 を定義して、26 行目で代入している。その後、v1 の要素の値を書き換えた後に v1, v2 の内容を表示している。実行結果が次になる。

```
$ ./6-12
dim: 3
0 0 0
Adrs: 0x7ffc0ebab4d0

dim: 3
0 0 0
Adrs: 0x7ffc0ebab4e0

dim: 3
1 2 3
Adrs: 0x7ffc0ebab4d0

dim: 3
1 2 3
Adrs: 0x7ffc0ebab4e0
```

2 回目に内容を表示した時に、v1 の要素を変更したのに、v2 の値も同じ値になってしまっている。これは代入処理の時に、配列の内容ではなく、先頭アドレスだけがコピーされたために v2 の方でも同じ領域を指し示しているからである。コピーコンストラクタの時と同じ問題が生じている。

次に示すのは、代入演算子 = にクラス myvector 用の専用の処理、具体的には動的に確保した配列の内容を適切に複製する処理を指定したプログラムである。

リスト 6.13　6-13.cpp

```
// g++ -Wall -ansi -std=c++11 -o 6-13 6-13.cpp
#include         <iostream>
#include         <cstdlib>

using namespace std;

const float V0[] = {0, 0, 0};

class myvector {
private:
    int                 dim;
    float         *ary;
public:
    myvector(const int dnum = 3, const float a[] = V0);
    myvector(const myvector& org);
    ~myvector();
    void operator = (const myvector& org);
    void print();
    void setvec(const float a[]);
    void getvec(float a[]);
};

int main(void)
{
    myvector        v1, v2;
    v2 = v1;
    v1.print();
    v2.print();

    float a[] = {1, 2, 3};
    v1.setvec(a);
    v1.print();
    v2.print();
/*
    myvector        v3;
    v3 = v2 = v1; // 連続した代入はまだ対応できない
    v1.print();
    v2.print();
    v3.print();
*/
    exit(EXIT_SUCCESS);
}

myvector::myvector(const int dnum, const float a[])
{
    dim = dnum;
```

(次のページに続く)

(前のページからの続き)

```
47      ary = NULL;
48      if (0 < dim) {
49          ary = new float[dim];
50          for (int i = 0; i < dim; i++) ary[i] = a[i];
51      }
52      else {
53          dim = 0;
54          ary = NULL;
55      }
56  }
57
58  myvector::myvector(const myvector& org)
59  {
60      dim = org.dim;
61      ary = new float[dim];
62      for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
63  }
64
65  myvector::~myvector()
66  {
67      delete[] ary;
68  }
69
70  void myvector::operator = (const myvector& org)
71  {
72      dim = org.dim;
73      float* tmp = new float[dim];
74      delete[] ary;
75      ary = tmp;
76      for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
77  }
78
79  void myvector::print()
80  {
81      cout << "dim: " << dim << endl;
82      if (0 < dim) {
83          for (int i = 0; i < dim; i++) cout << ary[i] << " ";
84          cout << endl;
85      }
86      cout << "Adrs: " << this << endl << endl;
87  }
88
89  void myvector::setvec(const float a[])
90  {
91      for (int i = 0; i < dim; i++) ary[i] = a[i];
92  }
93
94  void myvector::getvec(float a[])
```

(次のページに続く)

(前のページからの続き)

```
95  {
96      for (int i = 0; i < dim; i++) a[i] = ary[i];
97  }
```

演算子の再定義、オーバーロードは、18 行目にプロトタイプ宣言がある。書式は通常のメンバ関数と同じ形式になるが、関数名の代わりに、operator 演算子 () となる。また定義の実体は 71 行目からある。代入演算子 = の左辺が呼び出し元のオブジェクト、右辺が引数とされる。例として、

```
myvector a, b;
a = b;
```

であれば、a が呼び出し元 (this で参照可能)、b が引数 org となる。定義の内容は、73 行目で次元数のコピー、74 行目で新しい配列の確保、75 行目で古い配列の解放、76 行目で配列の先頭アドレスのコピー、77 行目で配列の内容のコピーをしている。新しく配列を確保し直すのは、次元数が異なっても正しく処理できるようにするためである。よって、この実装では次元数が異なるオブジェクト間でも代入が可能である。実行結果を次に示す。

```
$ ./6-13
dim: 3
0 0 0
Adrs: 0x7ffd9171b700

dim: 3
0 0 0
Adrs: 0x7ffd9171b710

dim: 3
1 2 3
Adrs: 0x7ffd9171b700

dim: 3
0 0 0
Adrs: 0x7ffd9171b710
```

6-12.cpp では、v1 の配列を変更した時に v2 も値が変わったように見えたが、この結果では v1 の配列を変更しても v2 には影響しない、すなわち正しく配列がコピーされていることがわかる。

ここまでで単一の代入は正しく処理できるようになった。しかし、37 行目でコメントアウトされているような連続した代入はまだできない。この部分を実行しようとすると、コンパイル時にエラーとなってしまう。

連続した代入が可能となるように代入処理を拡張すると、次に示す 6-15.cpp となる。6-13.cpp との違いは、void operator = () から myvector& operator = () のように、返り値として代入されたオブジェクトの参照を返すようにした点である。この変更により、右から順に代入処理を進めたときに、2 回目の代入時に代入すべきオブジェクトが存在することになる。77 行目の return が変更点である。

リスト 6.14　6-15.cpp

```
// g++ -Wall -ansi -std=c++11 -o 6-15 6-15.cpp
#include         <iostream>
#include         <cstdlib>

using namespace std;

const float V0[] = {0, 0, 0};

class myvector {
private:
    int                 dim;
    float         *ary;
public:
    myvector(const int dnum = 3, const float a[] = V0);
    myvector(const myvector& org);
    ~myvector();
    myvector& operator = (const myvector& org);
    void print();
    void setvec(const float a[]);
    void getvec(float a[]);
};

int main(void)
{
    float a[] = {1, 1, 1};
    myvector        v1(3, a), v2, v3;

    v1.print();
    v2.print();
    v3.print();

    a[1] = 2; a[2] = 3;
    v1.setvec(a);
    v3 = v2 = v1;

    v1.print();
    v2.print();
    v3.print();

    exit(EXIT_SUCCESS);
}

myvector::myvector(const int dnum, const float a[])
{
    dim = dnum;
    ary = NULL;
```

(次のページに続く)

(前のページからの続き)

```
    if (0 < dim) {
        ary = new float[dim];
        for (int i = 0; i < dim; i++) ary[i] = a[i];
    }
    else {
        dim = 0;
        ary = NULL;
    }
}

myvector::myvector(const myvector& org)
{
    dim = org.dim;
    ary = new float[dim];
    for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
}

myvector::~myvector()
{
    delete[] ary;
}

myvector& myvector::operator = (const myvector& org)
{
    dim = org.dim;
    float* tmp = new float[dim];
    delete[] ary;
    ary = tmp;
    for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
    return *this;
}

void myvector::print()
{
    cout << "dim: " << dim << endl;
    if (0 < dim) {
        for (int i = 0; i < dim; i++) cout << ary[i] << " ";
        cout << endl;
    }
    cout << "Adrs: " << this << endl << endl;
}

void myvector::setvec(const float a[])
{
    for (int i = 0; i < dim; i++) ary[i] = a[i];
}

void myvector::getvec(float a[])
```

(次のページに続く)

(前のページからの続き)

```
{
    for (int i = 0; i < dim; i++) a[i] = ary[i];
}
```

実行結果を次に示す。連続した代入の結果が正しく得られていることがわかる。

```
$ ./6-15
dim: 3
1 1 1
Adrs: 0x7ffc94e8a6e0

dim: 3
0 0 0
Adrs: 0x7ffc94e8a6f0

dim: 3
0 0 0
Adrs: 0x7ffc94e8a700

dim: 3
1 2 3
Adrs: 0x7ffc94e8a6e0

dim: 3
1 2 3
Adrs: 0x7ffc94e8a6f0

dim: 3
1 2 3
Adrs: 0x7ffc94e8a700
```

次に示すのは、もう一つの例としてクラス myvector の加算を演算子 + で実行できるようにする定義である。プロトタイプ宣言が 19, 20 行、実体の定義が 92 行から 121 行目となる。

リスト 6.15　6-16.cpp

```
// g++ -Wall -ansi -std=c++11 -o 6-16 6-16.cpp
#include         <iostream>
#include         <cstdlib>

using namespace std;

const float V0[] = {0, 0, 0};

class myvector {
private:
    int                 dim;
```

(次のページに続く)

(前のページからの続き)

```
    float        *ary;
public:
    myvector(const int dnum = 3, const float a[] = V0);
    myvector(const myvector& org);
    ~myvector();
    myvector& operator = (const myvector& org);
    myvector operator + (const myvector& rop);
    myvector operator + (const float f);
    void print();
    void setvec(const float a[]);
    void getvec(float a[]);
};

int main(void)
{
  {
    float a[] = {1, 1, 1};
    myvector        v1(3, a), v2, v3;

    v1.print();
    v2.print();
    v3.print();

    a[1] = 2; a[2] = 3;
    v1.setvec(a);
    v3 = v2 = v1;

    v1.print();
    v2.print();
    v3.print();

    v3 = v2 + v1;
//  v3.operator=(v2.operator+(v1)); 前行の式はこのように書くのと等価
    v3.print();

    v3 = v3 + 10; // 今回の実装では v3 = 10 + v3; とは書けないので注意する
    v3.print();
  }
    exit(EXIT_SUCCESS);
}

myvector::myvector(const int dnum, const float a[])
{
    cout << "Constructor called, " << this << endl;
    dim = dnum;
    ary = NULL;
    if (0 < dim) {
        ary = new float[dim];
```

(次のページに続く)

(前のページからの続き)

```
60          for (int i = 0; i < dim; i++) ary[i] = a[i];
61      }
62      else {
63          dim = 0;
64          ary = NULL;
65      }
66  }
67
68  myvector::myvector(const myvector& org)
69  {
70      cout << "Copy Constructor called, " << this << endl;
71      dim = org.dim;
72      ary = new float[dim];
73      for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
74  }
75
76  myvector::~myvector()
77  {
78      delete[] ary;
79      cout << "Destructor called, " << this << endl;
80  }
81
82  myvector& myvector::operator = (const myvector& org)
83  {
84      dim = org.dim;
85      float* tmp = new float[dim];
86      delete[] ary;
87      ary = tmp;
88      for (int i = 0; i < dim; i++) ary[i] = org.ary[i];
89      return *this;
90  }
91
92  myvector myvector::operator + (const myvector& rop)
93  {
94      const int MAXDIM = 10;
95
96      if (dim != rop.dim) {
97          cerr << "Error: dimension not match." << endl;
98          exit(EXIT_FAILURE);
99      }
100     if (MAXDIM < dim) {
101         cerr << "Error: Too large dimension." << endl;
102         exit(EXIT_FAILURE);
103     }
104
105     float a[MAXDIM];
106     for (int i = 0; i < dim; i++) a[i] = ary[i] + rop.ary[i];
107     return myvector(dim, a);
```

(次のページに続く)

(前のページからの続き)

```
}

myvector myvector::operator + (const float f)
{
    const int MAXDIM = 10;

    if (MAXDIM < dim) {
        cerr << "Error: Too large dimension." << endl;
        exit(EXIT_FAILURE);
    }
    float a[MAXDIM];
    for (int i = 0; i < dim; i++) a[i] = ary[i] + f;
    return myvector(dim, a);
}

void myvector::print()
{
    cout << "dim: " << dim << endl;
    if (0 < dim) {
        for (int i = 0; i < dim; i++) cout << ary[i] << " ";
        cout << endl;
    }
    cout << "Adrs: " << this << endl << endl;
}

void myvector::setvec(const float a[])
{
    for (int i = 0; i < dim; i++) ary[i] = a[i];
}

void myvector::getvec(float a[])
{
    for (int i = 0; i < dim; i++) a[i] = ary[i];
}
```

実行結果を以下に示す。

```
$ ./6-16
Constructor called, 0x7ffc9ae68490
Constructor called, 0x7ffc9ae684a0
Constructor called, 0x7ffc9ae684b0
dim: 3
1 1 1
Adrs: 0x7ffc9ae68490

dim: 3
0 0 0
```

(次のページに続く)

(前のページからの続き)

```
Adrs: 0x7ffc9ae684a0

dim: 3
0 0 0
Adrs: 0x7ffc9ae684b0

dim: 3
1 2 3
Adrs: 0x7ffc9ae68490

dim: 3
1 2 3
Adrs: 0x7ffc9ae684a0

dim: 3
1 2 3
Adrs: 0x7ffc9ae684b0

Constructor called, 0x7ffc9ae684c0
Destructor called, 0x7ffc9ae684c0
dim: 3
2 4 6
Adrs: 0x7ffc9ae684b0

Constructor called, 0x7ffc9ae684c0
Destructor called, 0x7ffc9ae684c0
dim: 3
12 14 16
Adrs: 0x7ffc9ae684b0

Destructor called, 0x7ffc9ae684b0
Destructor called, 0x7ffc9ae684a0
Destructor called, 0x7ffc9ae68490
```

6.11 まとめ

この章の目標

- オブジェクト指向を理解する
- class の記述の仕方を理解する
- private, public を適切に指定して、カプセル化を実現できる。

練習問題

1. 6-2.cpp で、38 行目の /*、47 行目の */ を削除して、実行してみなさい。

2. 前問に続けて class human の定義中の private を public に変更して、実行してみなさい。
3. メンバ変数 name を操作する、setname(), getname() を作成して動作を確認しなさい。間違いを防ぐチェック機能を組み込みなさい。(e6-34567.cpp)
4. メンバ変数 nation を操作する、setnation(), getnation() を作成して動作を確認しなさい。間違いを防ぐチェック機能を組み込みなさい。(e6-34567.cpp)
5. メンバ変数 age を操作する、setage(), getage() を作成して動作を確認しなさい。間違いを防ぐチェック機能を組み込みなさい。(e6-34567.cpp)
6. メンバ変数 height を操作する、setheight(), getheight() を作成して動作を確認しなさい。間違いを防ぐチェック機能を組み込みなさい。(e6-34567.cpp)
7. メンバ変数 weight を操作する、setweight(), getweight() を作成して動作を確認しなさい。間違いを防ぐチェック機能を組み込みなさい。(e6-34567.cpp)
8. 問題
9. 問題
10. 問題
11. n 次元ベクタを表す class myvector をコンストラクタとデストラクタを含めて実装しなさい。メンバ変数は次元を表す int dim; と各次元の要素を保持する配列のアドレスを保持する float ＊ ary; とする。配列は dim の値に基づき実行時に動的に確保しなさい。(e6-11121314.cpp)
12. n 次元ベクタを定数倍するメンバ関数 void mulvec(float v); を作成しなさい。(e6-11121314.cpp)
13. class myvector に対して乗算の演算子 ＊ の機能を拡張、オーバーロードして、n 次元ベクタを定数倍できるようにしなさい。(e6-11121314.cpp)
14. class myvector に対して減算の演算子 - の機能を拡張、オーバーロードして、n 次元ベクタ間で減算ができるようにしなさい。(e6-11121314.cpp)
15. 問題
16. 問題
17. 問題

第 7 章

オブジェクト指向 2、継承

7.1 継承

class には階層的な概念をデータとして適切に表現できるようにする**継承 (inheritance, inherit)** と呼ばれる機能が言語仕様として用意されている。上位概念を表現する class1 を定義しておいて、class1 に基づく下位概念 class 2 を定義する時に、class1 を継承すると指定しておくと class1 に含まれるメンバ変数やメンバ関数を自動的に class2 に持たせることができる。

上位概念のクラスを **基底クラス** 、下位概念のクラスを **派生クラス**と呼ぶ。また **親クラス** 、 **子クラス** と呼ぶこともある。継承により、クラス間の上位/下位、親子関係を表現できる。

これは現実世界における知識の表現と同じ原理、考え方となっている。例として、「動物はものを食べる」、「動物は子孫を作る」、などの知識を上位概念として持っていて、「哺乳類は動物である」、「犬は哺乳類である」、「ヒトは哺乳類である」などの知識が与えられれば、「犬はものを食べるか？」「ヒトはものを食べるか」などの質問に容易に答えることができる。このとき、「○○が動物であれば、○○は動物の性質を備えている」という推論ができるからである。継承はこの推論をプログラムとして直接表現できる。継承により、「犬はものを食べる」、「ヒトはものを食べる」などの知識を個別に記述する必要がなくなる。

以下に示す 7-1.h, 7-1.cpp は、前章のヒトを表す class のサンプルプログラム 6-2.cpp をもとにして、継承の機能を利用して動物が持つ性質を次の図のように階層的に表現する。

動物
↓
哺乳類、鳥、爬虫類、魚、昆虫
↓
ヒト

リスト 7.1　7-1.h

```
1 // g++ -Wall -ansi -std=c++11 -o 7-1 7-1.cpp
2 #include         <iostream>
3 #include         <cstdlib>
```

(次のページに続く)

(前のページからの続き)

```
4
5  const int         NO      = 0;
6  const int         YES     = 1;
7
8  const int         MALE    = 0;
9  const int         FEMALE  = 1;
10
11 const int         JAPAN   = 0;
12 const int         US      = 1;
13
14 using namespace std;
15
16 class animal {
17 protected:
18     int live;         // 生きている
19     int eat;          // ものを食べる
20     string move;      // 移動する
21     string temp;      // 体温
22     string breed   ; // 繁殖
23 public:
24     animal(){cout << "Constructor animal" << endl; live = YES; eat = YES;}
25     ~animal(){cout << "Destructor animal" << endl;}
26     void fa();
27     void print();
28 };
29
30 class insect : public animal { // 昆虫
31 public:
32     insect(){move = "6legs,wing"; temp = "climacteric"; breed = "egg";}
33 //  ~insect();                          // ^^e2^^ac^^86  変温動物
34 };
35
36 class fish : public animal {
37 public:
38     fish(){move = "fin"; temp = "climacteric"; breed = "egg";}
39 //  ~fish();
40 };
41
42 class reptiles : public animal { // 鳥
43 public:
44     reptiles(){move = "4legs"; temp = "climacteric"; breed = "egg";}
45 //  ~reptiles();
46 };
47
48 class bird : public animal {
49 public:
50     bird(){move = "2legs,wing"; temp = "constant"; breed = "egg";}
51 //  ~bird();                            // ^^e2^^ac^^86  恒温動物
```

(次のページに続く)

(前のページからの続き)

```
};

class mammalian : public animal { // 哺乳類、animal の派生クラス
public:
    mammalian(){cout << "Constructor mammalian" << endl;
                move = "4legs"; temp = "constant"; breed = "child,milk";}
    ~mammalian(){cout << "Destructor mammalian" << endl;}
    void fm();
//  void print(); // 現 class 用の関数を作成しないと基底クラスのものが使われる
};

class human : public mammalian { // ヒト、mammalian の派生クラス
private:
    string         name;
    int          nation; // nationality 国籍
    int                 sex;    // MALE or FEMALE
    int                 age;
    float        height; // meter
    float        weight; // kilo gram
public:
    human(string str = "noname", int n = 0, int s = 0, int a = 0,
          float h = 0, float w = 0);
    ~human(){cout << "Destructor human" << endl;}
    void set(string name, int n, int s, int a, float h, float w);
    void fh();
    void print(); // 現 class 用の関数を作成するとそれが使われる
};
```

リスト 7.2　7-1.cpp

```
// g++ -Wall -ansi -std=c++11 -o 7-1 7-1.cpp
#include         <iostream>
#include         <cstdlib>
#include         "7-1.h"

using namespace std;

int main(void)
{
  {
    animal        a;
    mammalian        m;
    human        h("Chubu Taro", JAPAN, MALE, 20, 1.75, 62.0);

    a.print();
    m.print(); // この場合は animal::print() が呼ばれる
    h.print(); // human::print() が呼ばれる

```

(次のページに続く)

(前のページからの続き)

```
    h.fa(); // human からでも animal::fa() を呼び出し可能
    h.fm();
    h.fh();
  }
    exit(EXIT_SUCCESS);
}

void animal::fa()
{
    cout << "member function animal" << endl;
}

void animal::print()
{
    cout << "live:" << live
         << "  eat:" << eat << endl;
}

void mammalian::fm()
{
    cout << "member function mammalian" << endl;
}
/*
void mammalian::print() // mammalian 用の print() を定義するとこれが使われる
{
    cout << "live:" << live
         << "  eat:" << eat
         << "  move:" << move
         << "  temp:" << temp
         << "  breed:" << breed << endl;
}
*/
human::human(string str, int n, int s, int a, float h, float w)
{
    cout << "Constructor human" << endl;
    name = str;
    nation = n;
    sex = s;
    age = a;
    height = h;
    weight = w;
}

void human::set(string str, int n, int s, int a, float h, float w)
{
// 略
}

```

(次のページに続く)

(前のページからの続き)

```
67 void human::fh()
68 {
69     cout << "member function human" << endl;
70 }
71
72 void human::print()
73 {
74     cout << "name:" << name
75          << "  nation:" << nation
76          << "  sex:" << sex
77          << "  age:" << age
78          << "  height:" << height
79          << "  weight:" << weight << endl;
80     cout << "live:" << live
81          << "  eat:" << eat
82          << "  move:" << move
83          << "  temp:" << temp
84          << "  breed:" << breed << endl;
85 }
```

7-1.h では

- 最上位の基底クラスの animal では、動物に共通するいくつかの性質をメンバ変数として定義している。コンストラクタでは、全体に共通する「生きている」、「ものを食べる」だけ設定する。他の変数は種類によって異なるので、ここでは設定していない。
- 2 番めの階層の動物の種類を表すクラスでは、種類固有の変数や関数は定義していないが、必要であれば定義できる。コンストラクタでは、メンバ変数にその種類固有の値を設定している。
- 最下位のクラス human は、mammalian の派生クラスとしている。これで、animal と mammalian のメンバ変数、関数を継承する。human 固有のメンバ変数、関数を追加している。

7-1.cpp では

- animal, mammalian, human のインスタンスを 1 個ずつ生成する。
- それぞれの print() を実行する。m.print() は実際は a.print() が実行される。
- h.fa(), h.fm(), h.fh() を実行する。それぞれは animal, mammalian, human のメンバ関数であるが、派生クラスからは基底クラスの関数をそのまま呼び出すことができる。

以下は実行結果である。コンストラクタとデストラクタがどのような順番で呼び出されるかがわかる。例として、human のインスタンス生成時は、animal, mammalian, human の順番で呼び出される。print() がどのように実行されるかもわかる。派生クラスで関数を定義しない場合は基底クラスの関数が呼び出される。

```
$ ./7-1
Constructor animal
```

(次のページに続く)

(前のページからの続き)

```
Constructor animal
Constructor mammalian
Constructor animal
Constructor mammalian
Constructor human
live:1  eat:1
live:1  eat:1
name:Chubu Taro  nation:0  sex:0  age:20  height:1.75  weight:62
live:1  eat:1  move:4legs  temp:constant  breed:child,milk
member function animal
member function mammalian
member function human
Destructor human
Destructor mammalian
Destructor animal
Destructor mammalian
Destructor animal
Destructor animal
```

以下は継承に関する新しい機能や記述、実行時の性質である。

7.2 アクセス指定子 protected

7-1.h 18 行目では、メンバ変数、メンバ関数のアクセスの範囲を指定するアクセス指定子として第 3 の指定子 protected が新たに使われている。

public はオブジェクト外からのアクセスを許可していて、private はオブジェクト外からのアクセスは許可しない。protected はオブジェクト外からは派生クラスだけアクセスを許可する。これにより、基底クラスと関連があり同じメンバ変数や関数を持つ派生クラスはそれらのメンバ関数、メンバ変数を継承できる。

7.3 継承するクラスの記述

継承するクラスは、7-1.h 31 行目のように記述する。一般形は次のようになる。

```
class 派生クラス名 : public 基底クラス名 {
...
}
```

基底クラス名の前にある public は継承したメンバ変数、関数のアクセス指定を表す。ここを public にすると、基底クラスで指定された変数や関数のアクセス指定を派生クラスでもそのまま継承する。すなわち、public のものは public に、private のものは private になる。一方、基底クラス名の前の指定を private にすると、もとのアクセス指定に関わらずすべて private になり、それ以降の派生クラスではアクセスできなくなる。この機能により、どの段階の派生クラスにまでアクセスを許可するか指定、制限することができる。通常は public とされることが多い。

7.4 関数のオーバーライド

関数のオーバーライド　とは、基底クラスに存在するメンバ関数と同じ名前の関数を派生クラスで再定義することである。7-1.cpp では、animal::print() に対する human::print() が相当する。関数のオーバーライドを行うと派生クラスでは再定義した方の関数が実行される。関数のオーバーライドを行わない場合は、基底クラスの関数が実行される。7-1.cpp では、mammalian::print() がコメントアウトされているが、この状態で mammalian のオブジェクトから print() を実行すると、animal::print() が実行される。

関数のオーバーライドを行うと、個々のクラスに適した処理を必要に応じて用意することができる。

5.6 節で出てきた、関数のオーバーロードとよく似た名称であるが、内容が異なるので混同しないように注意する。

- 関数のオーバーロードは、同じ名前で複数の処理を用意しておいて、その中から適切なものを選択して実行する。
- 関数のオーバーライドは、基底クラスに存在する関数を派生クラスの関数で上書きする、置き換えるような処理になる。

7.5 コンストラクタ、デストラクタが実行される順番

継承により親子関係があるクラスのオブジェクトが生成される場合、コンストラクタ、デストラクタが実行される順番は次のようになる。

- コンストラクタは、基底クラス → 派生クラスの順に呼ばれる。
- デストラクタは逆に、派生クラス → 基底クラスの順に呼ばれる。

例として、7-1.cpp で human h; としてオブジェクトを生成すると、h に対して、まず animal() が実行され、続いて mammalian() が実行され、最後に human() が実行される。基底クラスは派生クラスに何があるかを知らないのに対して、派生クラスは自分の元になっている基底クラスの構成をしっているので、場合によって派生クラスからは基底クラスのメンバ変数や関数を利用する可能性があるからである。

オブジェクトを破棄する時は逆の順番になる。基底クラス（の変数や関数）を先に破棄してしまうと、派生クラスのデストラクタが基底クラスの変数や関数をアクセスした時にエラーとなる。下位の派生クラスの破棄の処理が終わった段階で一段上の基底クラスを破棄することで問題が生じなくなる。

7.6 まとめ

この章の目標

- 継承の考え方を理解して、プログラム作成時に適切に活用できる。
- 目標 2
- 目標 3
- 目標 4

練習問題

1. 動物を表す 7-1.h, 7-1.cpp に別の種類の動物 (犬、猫など) を表す派生クラスを追加する。その動物に特有のメンバ変数、メンバ関数を定義して、設定や表示、実行できることを確認する。(e7-1234.h, e7-1234.cpp)
2. 両生類を表す派生クラスを追加する。その類に特有のメンバ変数、メンバ関数を定義して、設定や表示、実行できることを確認する。(e7-1234.h, e7-1234.cpp)
3. 派生クラスとして蛙など、両生類に分類される種類を追加する。(e7-1234.h, e7-1234.cpp)
4. class animal に鳴き声を表すメンバ変数を作成して、派生クラスの種で適切な値を設定する。(e7-1234.h, e7-1234.cpp)
5. 問題
6. 問題
7. 問題
8. 問題

第 8 章

標準テンプレートライブラリ、STL : Standard Template Library

8.1 標準テンプレートライブラリ とは何か

最初に C 言語で提供されるライブラリ関数の機能を見てみよう。C 言語では文字列処理の一連の関数群が利用できる。次のプログラムはそのような文字列を処理する関数のいくつかを利用している。

リスト 8.1　8-1.c

```c
// gcc -Wall -ansi -std=c99 -o 8-1 8-1.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define         SIZE        64

int main(void)
{
    int         i;
    char str[SIZE] = "abcde";
    char str2[SIZE];

    printf("%s %d\n", str, (int)strlen(str)); // 文字列の長さを求める
    strcpy(str2, str);                        // 文字列をコピーする
    printf("%s\n", str2);
    printf("%s %s %d\n", str, str2,
           strcmp(str, str2));                // 文字列を比較する
    for (i = 0; i < strlen(str); i++) {
        str[i] -= 0x20;
    }
    printf("%s %s %d\n", str, str2, strcmp(str, str2));
    strcat(str, str2);                                // 文字列を連結する
    printf("%s\n", str);
```

(次のページに続く)

(前のページからの続き)

```
25
26      exit(EXIT_SUCCESS);
27  }
28
```

```
$ ./8-1
abcde 5
abcde
abcde abcde 0
ABCDE abcde -32
ABCDEabcde
```

これらの関数を利用すると、自分で関数を書かなくても文字列の長さを求めたり、文字列を操作、加工することができる。しかし、対象が文字列 (char の配列) に限定されており、使える関数の種類も限られている。

これに対して C++ で提供される**標準テンプレートライブラリ STL : Standard Template Library** とは、近年の C++ の仕様の一部として標準的に提供されるライブラリ機能のうち、特にテンプレート機能を活用して **処理対象のデータの型に依存しない** 形で定義、作成された様々なデータ構造とそれに付随するデータのアクセス方法、およびデータ処理の機能 (関数) の総称である。データ構造とデータ処理の機能は、それぞれ次の名称で表される。

- **コンテナ Container** : STL により提供されるデータ構造とそれをアクセスする機能を表す。データの入れておく場所と考えるとよい。
- **イテレータ Iterator** : コンテナのデータを順番に取り出す機能を表す。英語の iterate は繰り返すという意味を持ち、そこから繰り返し取り出す操作を指すと考えられる。
- **アルゴリズム Algorithm** : コンテナのデータに対して適用可能な処理、関数を表す。一般的にプログラミングの世界でアルゴリズムという語は、計算や処理の手順を表すが、STL においては具体的な処理や関数を表すので注意する。この章ではすべて後者の意味でアルゴリズムという語を使用する。

これらは、互いに連携させて使用することを想定して作成されている。また、データ構造と処理の組み合わせの **直交性** がなるべく保たれるように、コンテナ、イテレータ、アルゴリズムはそれぞれ設計、実装されている。

また次の C++ の機能は STL により提供されるものではないが、アルゴリズムの一部の関数で使用されるため、ここであわせて紹介しておく。

- **関数オブジェクト Function Object** : オブジェクト自身に関数の機能を持たせたものを表す。

4 項目それぞれの詳細はこの後の各節で取り上げる。

STL は名前空間 std で定義されている。実際に STL を活用してプログラムを作成する時は、using で std を指定することで、名前空間の指定 std:: を省略することができる。この後示すサンプルプログラムではどの部分が STL に関する記述かわかるように、using を指定せず、STL の機能はすべて std:: を付加した状態で記述する。

8.2 コンテナ

コンテナとは、STL により提供されるデータ構造、およびそれにより保持されるデータを指す。コンテナには、通常の配列と同様の、要素となるデータがメモリ上で連続的に並んだ状態で記憶されるものから、キュー、２分木など様々な種類がある。データ構造の種類に応じてデータをアクセスする方法も異なるものが用意される。

コンテナで用意、提供される代表的なデータ構造を以下に示す。

- array : 固定長配列
- vector : 可変長配列、要素数の変更が可能
- deque : Double Ended QUEue を省略した名称で、両端での操作が可能なキューを表す
- list : 双方向連結リスト、要素間で前後両方向に連結されている
- set : 順序付けられた集合
- map : 連想配列、順序付けられている

コンテナの機能の一部を利用して提供される **コンテナアダプタ** と呼ばれるデータ構造とそのためのアクセス方法もある。コンテナアダプタは STL で必要とされる仕様を満たしていない部分があるため、STL で提供される機能で使えないものがある。

次のプログラムはコンテナと次節で示すイテレータの使い方を示している。コンテナの例として、比較的使う機会が多いと予想される、可変長配列を実現する vector を取り上げる。STL の vector は 6 章の class のサンプルプログラムで取り上げた myvector とは別のものであるので混同しないように注意する。

リスト 8.2　8-2.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 8-2 8-2.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4  #include         <vector> // 可変長の配列を実現するコンテナ vector
5
6  const int SIZE = 10;
7
8  void print(const std::vector<int> &v);
9
10 int main(void)
11 {
12     std::vector<int> v; // int を要素とする vector 型のコンテナの宣言
13     std::vector<int> v2(SIZE);
14     std::vector<int> v3 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
15
16 // 初期状態
17
18     std::cout << v.size() << std::endl; // size(), コンテナの要素数を求める関数
19 //  std::cout << v.at(0) << std::endl;  // 要素がないとこれはエラー
```

(次のページに続く)

(前のページからの続き)

```
    std::cout << std::endl;

// 値の代入

    for (int i = 0; i < SIZE; i++) {
        v.push_back(i); // 配列の末尾に値 i を新要素として追加する
    }
    for (int i = 0; i < SIZE; i++) {
        v2.at(i) = i*10; // 配列の添字 i 番目の場所に値を書き込む
    }
    std::cout << v.size() << std::endl; // この時点の要素数
    std::cout << std::endl;

// イテレータ、要素を一つずつ取り出してきて処理する

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        std::cout << *it << " "; // 先頭から順に取り出す、ポインタと同じ形式
    }
    std::cout << std::endl;

    for (std::vector<int>::reverse_iterator rit = v.rbegin(); rit != v.rend(); rit++) {
        std::cout << *rit << " "; // 末尾から、逆順で取り出すこともできる
    }
    std::cout << std::endl;
    std::cout << std::endl;

// auto を指定するとイテレータの適切な型を推論させることが可能

    for (auto ai = v2.begin(); ai != v2.end(); ai++) {
        std::cout << *ai << " ";
    }
    std::cout << std::endl;

    print(v3); // 同じ表示の機能を関数化した
    std::cout << std::endl;

// 可変長の機能の確認

    v3.push_back(11); // 要素の追加
    v3.push_back(12);
    v3.push_back(13);
    std::cout << v3.size() << " : "; print(v3);

    v3.erase(v3.begin() + 3, v3.begin() + 7); // 要素を 4 個取り除く
    std::cout << v3.size() << " : "; print(v3);

    exit(EXIT_SUCCESS);
}
```

(次のページに続く)

(前のページからの続き)

```
68
69 void print(const std::vector<int> &v) // 引数として渡す時はこのように書く
70 {
71     for (auto i = v.begin(); i != v.end(); i++) {
72         std::cout << *i << " ";
73     }
74     std::cout << std::endl;
75 }
```

```
$ ./8-2
0

10

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

0 10 20 30 40 50 60 70 80 90
0 1 2 3 4 5 6 7 8 9

13 : 0 1 2 3 4 5 6 7 8 9 11 12 13
9 : 0 1 2 7 8 9 11 12 13
```

次のプログラムは int, float などの通常のデータ型でない class を対象としても vector 型のコンテナが作成、操作できることを示している。

リスト 8.3　8-3.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 8-3 8-3.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4  #include         <vector>
5
6  const int SIZE = 10; // 可変長の配列の要素数
7
8  class xyz { // 3次元の座標を表すクラス
9  private:
10     float x;
11     float y;
12     float z;
13 public:
14     xyz(float tx = 0, float ty = 0, float tz = 0){x = tx; y = ty; z = tz;}
15     float getx(){return x;}
16     float gety(){return y;}
17     float getz(){return z;}
18 };
19
```

(次のページに続く)

(前のページからの続き)

```
20 void print(std::vector<xyz> &v);
21
22 int main(void)
23 {
24     xyz tmp(1, 2, 3);
25     std::vector<xyz> v(SIZE); // xyz 型のオブジェクトを要素とする vector
26
27     for (int i = 0; i < SIZE; i++) {
28         v.at(i) = tmp; // 配列の添字 i 番目の場所に値を代入する
29     }
30 // 次のように書いてもよい、同じ結果になる
31 //  for (auto it = v.begin(); it != v.end(); it++) *it = tmp;
32
33     print(v);
34
35     exit(EXIT_SUCCESS);
36 }
37
38 void print(std::vector<xyz> &v) // const を外したのはメンバ関数のため
39 {
40     for (auto i = v.begin(); i != v.end(); i++) {
41         std::cout << i -> getx() << " " <<
42                      i -> gety() << " " <<
43                      i -> getz() << std::endl;
44     }
45 }
```

```
$ ./8-3
1 2 3
1 2 3
1 2 3
.
.
1 2 3
```

8.3 イテレータ

イテレータは、コンテナに含まれるデータを順に取り出すために用意されている STL の機能である。イテレータにより取り出されたデータに、必要な処理を施して結果を得るのが一般的な処理の流れとなる。

vector と組み合わせて使われるイテレータは、配列の先頭から最後尾に向かって順に要素を取り出すものと、最後尾から先頭に向かって逆向きに取り出すものがある。

vector におけるイテレータの典型的な使い方は以下のようになる。

```
for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    // *it で要素を参照できるので、ここに必要な処理を記述する
}
```

繰り返し範囲の終端を表す end() は最後に処理するデータの次の要素を指すので注意する。これは for ループを記述する時に、10 回の繰り返しであれば、

```
for (int i = 0; i < 10; i++) {

}
```

と書くのと同じ考え方になる。このループ内部では、i は 0 から 9 まで変化するが 10 の時は処理されずループから抜けてしまう。つまり処理対象の 0--9 に対して、その次の値を終了条件に記述している。

8.4 アルゴリズム

アルゴリズムは、STL により提供されるデータ処理の関数を指す。アルゴリズムの各関数はコンテナやイテレータと組み合わせて使うことで、数多くの処理を簡潔な記述で実行できるように設計、作成されている。アルゴリズムには 100 種類を超える関数が用意されている。

次のプログラムは vector 型のコンテナに対して使用できる代表的なアルゴリズムの関数の使い方を示している。

リスト 8.4　8-4.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 8-4 8-4.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4  #include         <vector>
5  #include         <algorithm> // アルゴリズムを使うときはこれが必要
6
7  const int SIZE = 10;
8
9  void print(const std::vector<int> &v);
10
11 int main(void)
12 {
13     std::vector<int> v(SIZE), v2(SIZE), v3(SIZE), v4(SIZE);
14
15 // アルゴリズムを使うことで、コンテナに対して様々な処理ができる
16
17 // fill(), 指定した値を配列の指定した範囲の要素に代入する
18     std::fill(v.begin(), v.end(), 0); // v 全体に 0 を代入する
19     print(v);
20     std::fill(v.begin(), v.begin() + v.size(), 1); // v 全体に 1 を代入する
21     print(v);
22     std::fill_n(v.begin() + 1, 5, 2); // v の途中の範囲に 2 を代入する
```

(次のページに続く)

(前のページからの続き)

```
    print(v);
    std::fill_n(v.begin(), v.size(), 3); // v 全体に 3 を代入する
    print(v);
    std::cout << std::endl;

// copy(), 領域のコピー
    std::fill(v.begin(), v.end(), 1); // 各コンテナの内容が区別できるように初期化
    std::fill(v2.begin(), v2.end(), 2);
    std::fill(v3.begin(), v3.end(), 3);
    std::fill(v4.begin(), v4.end(), 0);
    std::copy(v.begin(), v.end(), v4.begin()); // コピーの例 v → v4
    print(v4);
    std::copy(v2.begin(), v2.end() - 5, v4.begin() + 2);
    print(v4);
    std::copy_n(v3.begin(), 3, v4.begin() + 4);
    print(v4);
    std::cout << std::endl;

// swap(), 値の交換
    std::fill(v.begin(), v.end(), 1);
    std::fill(v4.begin(), v4.end(), 0);
    std::iter_swap(v.begin() + 8, v4.begin() + 8); // 1 個交換する
    std::swap_ranges(v.begin() + 1, v.end() - 5, v4.begin() + 1); // 複数個の交換
    print(v);
    print(v4);
    std::cout << std::endl;

// remove(), 指定した値を取り除く
    std::fill(v4.begin() + 5, v4.end() - 1, 2);
    print(v4);
  {
    auto result = std::remove(v4.begin(), v4.end(), 1); // 値 1 の要素を除く
    print(v4);
    std::cout << v4.size() << std::endl;
    v4.erase(result, v4.end()); // 必要な要素は先頭に集められるので、不要な部分を削除
  }
    print(v4); // 結果だけが残っているのがわかる
    std::cout << v4.size() << std::endl;
    std::cout << std::endl;

// replace(), 指定した値を別の値で置き換える
    v4.push_back(2);
    v4.push_back(3);
    v4.push_back(4);
    v4.push_back(5);
    print(v4);
    replace(v4.begin(), v4.end(), 2, 7); // 値 2 の要素を 7 に書き換える
    print(v4);
```

(次のページに続く)

(前のページからの続き)

```
    std::cout << std::endl;

// sort(), ソートする
    std::sort(v4.begin(), v4.end()); // 値の小さい順に並べ替える
    print(v4);

// unique(), ソート済の配列から重複した値を取り除く
  {
    auto result = std::unique(v4.begin(), v4.end());
    print(v4);

// 不要な値を削除する
    v4.erase(result, v4.end());
  }
    print(v4);
    std::cout << v4.size() << std::endl;
    std::cout << std::endl;

    exit(EXIT_SUCCESS);
}

void print(const std::vector<int> &v)
{
    for (auto i = v.begin(); i != v.end(); i++) {
      std::cout << *i << " ";
    }
    std::cout << std::endl;
}
```

```
$ ./8-4
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
1 2 2 2 2 2 1 1 1 1
3 3 3 3 3 3 3 3 3 3

1 1 1 1 1 1 1 1 1 1
1 1 2 2 2 2 2 1 1 1
1 1 2 2 3 3 3 1 1 1

1 0 0 0 0 1 1 1 0 1
0 1 1 1 1 0 0 0 1 0

0 1 1 1 1 2 2 2 2 0
0 2 2 2 2 0 2 2 2 0
10
0 2 2 2 2 0
6
```

(次のページに続く)

(前のページからの続き)

```
0 2 2 2 2 0 2 3 4 5
0 7 7 7 7 0 7 3 4 5

0 0 3 4 5 7 7 7 7 7
0 3 4 5 7 7 7 7 7 7
0 3 4 5 7
5
```

範囲の終端を表す end() は、イテレータの場合と同じく、最後に処理するデータの次の要素を指すので注意する。

次のプログラムは vector と配列の親和性を示している。vector は通常の配列と同じ操作ができるように実装されているので、vector に対して配列の要素をアクセスする記法が受け付けられる。また vector に対応するアルゴリズムの関数は通常の配列に対しても使用することができる。

リスト 8.5　8-5.cpp

```
1  // g++ -Wall -ansi -std=c++11 -o 8-5 8-5.cpp
2  #include         <iostream>
3  #include         <cstdlib>
4  #include         <vector>
5  #include         <algorithm>
6
7  const int SIZE = 10;
8
9  void print(const std::vector<int> &v); // vector 型コンテナの表示
10 void print(int v[]);                    // 通常の配列の表示
11
12 int main(void)
13 {
14     int v1[SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
15     std::vector<int> v2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
16
17 // vector は普通の配列と同じアクセスの仕方で操作可能
18
19     v1[0] = 99;
20     v2[0] = 99; // vector もこのように書いてよい
21     v2.at(1) = 99;
22 //  v2[SIZE+1] = 99; // 範囲外でもエラーにならず実行できてしまう
23 //  v2.at(SIZE+1) = 99; // at() を使うと範囲外はエラーで検出できる
24     print(v1);
25     print(v2);
26
27 // アルゴリズムでは通常の配列も vector と同じように操作可能
28
29     std::fill(v1, v1 + 10, 1); // 通常の配列もこのようにして渡せる
30     print(v1);
31     std::fill(v2.begin(), v2.end(), 2);
```

(次のページに続く)

(前のページからの続き)

```
32      print(v2);
33      std::fill(&(v2[0]), &(v2[SIZE]), 3); // 無理やりポインタのように書いた
34      print(v2);
35
36      exit(EXIT_SUCCESS);
37  }
38
39  void print(const std::vector<int> &v)
40  {
41      for (auto i = v.begin(); i != v.end(); i++) {
42          std::cout << *i << " ";
43      }
44      std::cout << std::endl;
45  }
46
47  void print(int v[])
48  {
49      for (int i = 0; i < SIZE; i++) {
50          std::cout << v[i] << " ";
51      }
52      std::cout << std::endl;
53  }
```

```
$ ./8-5
99 1 2 3 4 5 6 7 8 9
99 99 2 3 4 5 6 7 8 9
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
```

8.5 関数オブジェクト

関数オブジェクトは、オブジェクトそのものに関数の機能を持たせたもの、関数と同様に呼び出せるオブジェクトのことを指す。関数オブジェクトそのものは STL により提供される機能ではないが、アルゴリズムの一部の機能が、関数オブジェクトにより処理内容を決められるように設計、実装されているため、STL と関係がある。関数オブジェクトを定義する方法はいくつかある。その一つとして、オブジェクトにおいて関数呼び出し演算子のオーバーロードを行う方法がある。この方法により定義される関数オブジェクトは **ファンクタ functor** と呼ばれる。

関数オブジェクトと同様の機能で、C 言語でも使える **関数ポインタ** という機能がある。関数オブジェクトは、オブジェクト内にメンバ変数を持つことができるので、内部状態を記憶しておいて、それにより動作を制御することが可能で、関数ポインタを使うよりもより複雑な処理を実現しやすい。

以下は簡単な関数オブジェクトの例である。

リスト 8.6　8-6.cpp

```
// g++ -Wall -ansi -std=c++11 -o 8-6 8-6.cpp
#include         <iostream>
#include         <cstdlib>

class xyz { // ３次元の座標を表すクラス
private:
    float x;
    float y;
    float z;
public:
    xyz(float tx = 0, float ty = 0, float tz = 0){x = tx; y = ty; z = tz;}
    void operator()() { std::cout << x << " " << y << " " << z << std::endl; }
//  関数呼び出し演算子 () をオーバーロードして必要な動作を実現する
};

int main(void)
{
    xyz tmp(1, 2, 3);

// tmp はオブジェクトであるが、() をオーバーロードして関数を呼び出す代わりに
// 変数を表示する機能を定義した。() を指定するとその動作をさせることができる。
    tmp();

    exit(EXIT_SUCCESS);
}
```

```
$ ./8-6
1 2 3
```

STL においてアルゴリズムの一部の関数は、関数オブジェクトにより動作を変更することができる。以下はその例で、ソートをする関数の大小関係の判定を関数オブジェクトにより変更している。class normal は小さい順に並べる判定をする functor 、class reverse は大さい順に並べる判定をする functor となる。

リスト 8.7　8-7.cpp

```
// g++ -Wall -ansi -std=c++11 -o 8-7 8-7.cpp
#include         <iostream>
#include         <cstdlib>
#include         <vector>
#include         <algorithm> // アルゴリズムを使うときはこれが必要

class normal {
public:
    bool operator()(int x, int y) { return x < y ? true : false; }
//  bool operator()(int x, int y) { return x < y }   // こうしても同じ結果になる
```

(次のページに続く)

(前のページからの続き)

```
};

class reverse {
public:
    bool operator()(int x, int y) { return y < x ? true : false; }
};

void print(const std::vector<int> &v);

int main(void)
{
    std::vector<int> v = {0, 1, 2, 8, 9, 3, 4, 5, 6, 7};
    normal n;
    reverse r;

    print(v);
    std::sort(v.begin(), v.end()); // 値の小さい順に並べ替える
    print(v);

    v.at(5) = 99;
    print(v);

    std::sort(v.begin(), v.end(), n); // 値の小さい順に並べ替える
    print(v);
    std::sort(v.begin(), v.end(), r); // 値の大きい順に並べ替える
    print(v);

    std::cout << n(1, 2) << " " << r(1, 2) << std::endl; // こんな呼び出し方もできる

    exit(EXIT_SUCCESS);
}

void print(const std::vector<int> &v)
{
    for (auto i = v.begin(); i != v.end(); i++) {
      std::cout << *i << " ";
    }
    std::cout << std::endl;
}
```

```
$ ./8-7
0 1 2 8 9 3 4 5 6 7
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 99 6 7 8 9
0 1 2 3 4 6 7 8 9 99
99 9 8 7 6 4 3 2 1 0
1 0
```

8.6 まとめ

この章の目標

- STL の概念、実現している機能を理解する。
- コンテナ、イテレータ、アルゴリズム、関数オブジェクトの概念、実現している機能を理解する。
- STL を活用してプログラムを作成できる。
- 目標
- 目標

練習問題

1. STL の機能を使わずに、通常の配列、for ループなどを組み合わせて、次の機能を持つプログラムを C++ で作成しなさい。10 個の要素を持つ int の配列 a[], b[] を定義する。a[] を 0 で初期化する。a[] を b[] にコピーする。b[] を画面に表示する。(e8-1.cpp)
2. 前記の問題を STL の機能、コンテナ、イテレータ、アルゴリズムを活用して作成しなさい。(e8-2.cpp)
3. 前記の問題を using を使って std の名前空間を指定して作成しなさい。(e8-3.cpp)
4. STL の機能を使わずに、通常の配列、for ループなどを組み合わせて、次の機能を持つプログラムを C++ で作成しなさい。int ＊ a に対して 10 個の要素を持つ配列を動的に確保して 0 で初期化する。int ＊ b に対して 11 個の要素を持つ配列を動的に確保して配列 a を配列 b にコピーする。b の最後の要素に 0 を代入する。a の領域を解放する。b の内容を画面に表示する。(e8-4.cpp)
5. 前記の問題を STL の機能、コンテナ、イテレータ、アルゴリズムを活用して作成しなさい。(e8-5.cpp)
6. 問題
7. 問題

第 9 章

総合演習課題

これまで取り上げた内容を理解してプログラム作成時に活用できるか、練習問題を通して確認する。プログラム作成に際して、以下を共通の条件とする。

- 変数の初期値の指定がされていない場合は適当な値を代入しておく。
- main() から呼び出す、と指示されている場合は、main() を含めてコンパイルに必要な記述を一通り含める。実行時に結果の表示ができるようにする。

9.1 C++ の基本

1. 整数を 1 個 KBD (キーボード) から入力して、画面に表示する。cin, cout を使用する。(e91-1.cpp)
2. 文字列を 1 個 KBD から入力して、画面に表示する。cin, cout を使用する。(e91-2.cpp)
3. 上記のプログラムを using を指定して作成する。(e91-3.cpp)
4. 2 個の整数の加算を行い返り値とするグローバル関数 int f1(int a, int b) を作成する。名前空間 n1 を定義し、その中で 2 個の整数の減算を行い返り値とする関数 int f1(int a, int b) を作成する。main() から両者を呼び出す。(e91-4.cpp)
5. 上記のプログラムを n1 に対して using を指定して作成する。(e91-5.cpp)
6. 引数 a, b, c を小さい順に並び替える関数 void f2(int ＊ a, int ＊ b, int ＊ c) をポインタを使って作成する。並び替えた結果を小さい順に a, b, c に代入する。main() から呼び出す。(e91-6.cpp)
7. 引数 a, b, c を小さい順に並び替える関数 void f3(int &a, int &b, int &c) を参照を使って作成する。並び替えた結果を小さい順に a, b, c に代入する。main() から呼び出す。(e91-7.cpp)
8. new を使って int 10 個分の配列を動的に確保する。配列の要素に 1 から順に 1 ずつ増える値を代入して画面に表示する。その後配列を delete で解放する。(e91-8.cpp)
9. const を使って int の整数定数を 1 個定義し、画面に出力する。(e91-9.cpp)

10. int の配列中の最小値を返り値とする関数 int f4(int a[], int n) を作成する。float の配列中の最小値を返り値とする関数 float f4(float a[], int n) を作成する。n は配列の要素数を表す。両者を main() から呼び出す。(e91-10.cpp)

11. 上記のプログラムを関数のテンプレートを利用して作成する。(e91-11.cpp)

9.2 class

前半の問題の機能をまとめて book.cpp として作成する。後半の問題の機能をまとめて complex.cpp として作成する。

1. 本を表すクラス book を定義する。メンバ変数として string title; string author; int year; int pages; を定義する。メンバ変数は private, メンバ関数は public のアクセス指定とする。(book.cpp)

2. 上記の book のコンストラクタを作成する。デフォルト引数として string 型は "none", int は 0 を代入する。main() から呼び出す。

3. 上記の book のデストラクタを作成する。本のタイトルと破棄されたことを示すメッセージを表示する。main() から呼び出す。

4. 上記の book において、string title; の入出力を行うメンバ関数 void SetTitle(string str), void GetTitle(string& str) を作成する。main() から呼び出す。

5. 上記の book において、メンバ関数 void print() を作成する。本の情報を表示する。メンバ変数の参照時は this ポインタを使用する。

6. 複素数を表すクラス mycomplex を定義する。メンバ変数として実部を表す float re; と虚部を表す float im; を定義する。メンバ変数は private, メンバ関数は public のアクセス指定とする。(complex.cpp)

7. 上記の mycomplex のコンストラクタを作成する。デフォルト引数として re, im に 0 を代入する。main() から呼び出す。

8. 上記の mycomplex において、re, im の入出力を行うメンバ関数 void set(float r, float i), void get(float& r, float& i) を作成する。main() から呼び出す。

9. 上記の mycomplex において、メンバ関数 void print() を作成する。"1 + 2i" のように表示する。

10. 上記の mycomplex において、演算子 = の動作を定義する。main() から呼び出す。

11. 上記の mycomplex において、演算子 + の動作を定義する。main() から呼び出す。

12. 上記の mycomplex において、演算子 ＊ の動作を定義する。main() から呼び出す。

9.3 継承

下記の機能をまとめて book2.cpp として作成する。

1. 前節の本を表すクラス book において、book を継承する派生クラスにだけメンバ変数を公開するようにアクセス指定を変更しなさい。(book2.cpp)
2. book の派生クラスとしてクラス dictionary を定義しなさい。メンバ変数として string language; を定義しなさい。
3. dictionary のコンストラクタを定義しなさい。language の初期値として "Japanese" を設定しなさい。
4. dictionary において language の入出力を行うメンバ関数 void SetLang(string l), void GetLang(string& l) を作成する。main() から呼び出す。
5. dictionary のメンバ変数をすべて表示するメンバ関数 print() を定義しなさい。main() から呼び出す。
6. book の派生クラスとしてクラス magazine を定義しなさい。メンバ変数として int month; を定義しなさい。
7. magazine のコンストラクタを定義しなさい。month の初期値として 0 を設定しなさい。
8. magazine において month の入出力を行うメンバ関数 void SetMonth(int m), void GetMonth(int& m) を作成する。main() から呼び出す。
9. magazine のメンバ変数をすべて表示するメンバ関数 print() を定義しなさい。main() から呼び出す。
10. book の派生クラスとしてクラス textbook を定義しなさい。メンバ変数として string subject; を定義しなさい。
11. textbook のコンストラクタを定義しなさい。subject の初期値として "Robotics" を設定しなさい。
12. textbook において subject の入出力を行うメンバ関数 void SetSbj(string s), void GetSbj(string& s) を作成する。main() から呼び出す。
13. textbook のメンバ変数をすべて表示するメンバ関数 print() を定義しなさい。main() から呼び出す。

9.4 STL

下記の機能をまとめて stl.cpp として作成する。

1. 10 個の int を要素とする vector 型のコンテナを作成しなさい。(stl.cpp)
2. イテレータを使用して、適当な値を代入しなさい。すべての要素を表示しなさい。
3. 要素を 3 個追加しなさい。すべての要素を表示しなさい。
4. 要素を 2 個削除しなさい。すべての要素を表示しなさい。
5. fill() を使用して配列の一部に同じ値を代入しなさい。すべての要素を表示しなさい。

6. replace() を使用して一部の値を変更しなさい。すべての要素を表示しなさい。

7. sort() を使用して値の小さい順に並べ替えなさい。すべての要素を表示しなさい。

第 10 章

練習問題の答え

準備中

第 11 章

マークアップのサンプル レベル１

11.1 マークアップのサンプル　レベル２

11.1.1 サンプル　レベル３

装飾のサンプル

太字、*イタリック*、

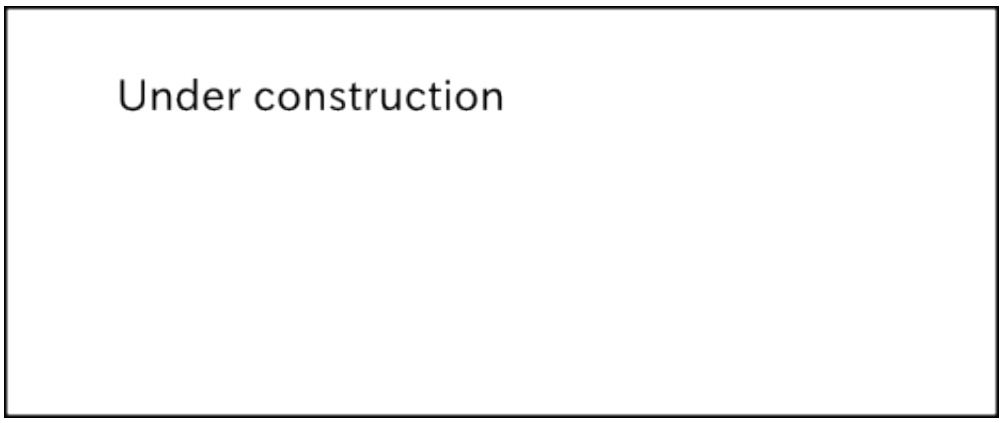

図 11.1　図のサンプル

本文中での参照は、図のサンプル 図 11.1 を御覧ください。

数式のサンプル $ABC + A\overline{B}C$

$$\overline{A}B\overline{C} + A\overline{B} + AC =$$
$$\overline{A}B\overline{C} + A\overline{B}(1) + A(1)C =$$

例題 3-2. 例題のサンプルを示しなさい。

答 3-2.　答のサンプル

リストのサンプル

箇条書き

- 順序回路の構成と動作原理が理解できる。
- ミーリー型順序回路とムーア型順序回路の違いがわかる。
- 状態遷移図、状態遷移表が理解できる、作成できる。
- 宇都宮市
- 那須塩原市
- 真岡市
- 2 次元座標、行列、分数
- 乗物、スポーツ、時間

数字付きリスト

1. まさし
2. みんみん
3. 夢餃子 (#を使うと、自動で数字が割り当てられます)

定義リスト

餃子　宇都宮の名物として有名。餃子の像もある。静岡の浜松がライバル。

ジャズ　宇都宮はジャズの町としても売り出し中。楽器メーカーを多数抱える静岡の浜松がライバル

焼きそば　知る人ぞ知る宇都宮の名物。専門店多数。なぜかビニール袋で持ち帰る。

表のサンプル

入力			出力					
A	0	1	A + 0	A + 1	A + A	A • 0	A • 1	A • A
0	0	1	0	1	0	0	0	0
1	0	1	1	1	1	0	1	1

これは **強調** のサンプルである。

例題 2-1. $(13)_{10}$ を 2 進数に変換しなさい。

答 2-1.

```
2 ) 13
  ------
2 )  6 ... 1
```

(次のページに続く)

(前のページからの続き)

```
  ------
2 )  3 ... 0
  ------
     1 ... 1
```

13 [10] で上付文字になる。

コードサンプルの表示

リスト 11.1　2-2.cpp

```
#include <cstdio>
#include <cstdlib>

class a {
    int a;
} b;

int main(void)
{
    a c;

    printf("Hello world.\n");

    exit(EXIT_SUCCESS);
//  exit(EXIT_FAILURE);
}
```

リスト 11.2　2-2.cpp

```
// g++ -Wall -ansi -std=c++11 -o 2-2 2-2.cpp
#include <cstdio>
#include <cstdlib>

int main(void)
{
    printf("Hello world. (2-2.cpp)\n");

    exit(EXIT_SUCCESS);
//  exit(EXIT_FAILURE);
}
```

rst ファイルを読み込む。

第 12 章

rst ファイルのサンプル

12.1 タイトル

箇条書きの例

- コンパイラの機能と使い方
- デバッグの方法、デバッガの使い方
- C, C++, プログラム作成の基本スキル
- C++ の様々な機能
- オブジェクト指向 1、class
- オブジェクト指向 2、継承
- 標準テンプレートライブラリ, STL : Standard Template Library
- 総合演習課題

12.2 まとめ

この章の目標

- このファイルで示す内容が理解できる。
- 様々なマークアップ言語から自分にあったものを選ぶことができる。

練習問題

1. rst ファイルの特徴を簡単に説明しなさい。
2. rst ファイルを自分で書いてみなさい。

文書ファイルをそのまま読み込む。

```
rst ファイルのサンプル
========================================

タイトル
----------------------------------------

箇条書きの例

- コンパイラの機能と使い方
- デバッグの方法、デバッガの使い方
- C, C++, プログラム作成の基本スキル
- C++ の様々な機能
- オブジェクト指向 1、class
- オブジェクト指向 2、継承
- 標準テンプレートライブラリ, STL : Standard Template Library
- 総合演習課題

まとめ
----------------------------------------

この章の目標

- このファイルで示す内容が理解できる。
- 様々なマークアップ言語から自分にあったものを選ぶことができる。

練習問題

#. rst ファイルの特徴を簡単に説明しなさい。
#. rst ファイルを自分で書いてみなさい。
```

第 13 章

Indices and tables

- genindex
- modindex
- search