プログラミング AII

(C プログラマのための C++プログラミング)

新潟大学工学部
知能情報システムプログラム２年第 2 ターム
協創経営プログラム　　３年第 2 ターム

新潟大学創生学部
知能情報システム領域学習科目パッケージ ２年第 2 ターム

平成 32 年 3 月 25 日

元木 達也
motoki@ie.niigata-u.ac.jp

目 次

0　ガイダンス

- 受講に当っての留意事項
- 授業の目標
- 教科書、参考書
- 授業予定

0.1　受講に当っての留意事項

旧カリキュラムにおける位置付け

- 平成 28 年度以前入学の情報工学科受講生においては、(合格したら) 「プログラミング I」に読み替えられる。

必要な予備知識

- この講義／実習は**プログラミングの入門コースではない**ので、1 年次第 3~4 タームの「プログラミング基礎 I, II」と前タームの「プログラミング AI」を既に履修して (ある程度の理解をして) いることを前提に話を進める。

授業の進め方

- 講義と演習／実習を交互に行う。$\Longrightarrow$ 基本的には、3 限は講義、4 限は演習／実習。
- C++プログラムの書き方の詳細は講義ノートや参考書等に書かれているので、**授業では細かい話はしない。**

⇨ **講義ノートや 参考書等を予め予習をして、授業を聞いても分からない部分は質問するようにして下さい。**

授業に出席するだけではこの授業の単位を取れないことを自覚して下さい。

0.2　達成目標

- **オブジェクト指向**の考え方を理解する。

すなわち、
ソフトウェア開発を職人芸から工業へ移行するためには、個々のソフトウェアモジュールを再利用可能なソフトウェア部品とする事が出来なければならない。そのためのプログラムパラダイムとして広く普及して来た**オブジェクト指向**の考え方を理解する。

そのために、
実際に **C++言語**を使ってオブジェクト指向プログラミングに慣れる。

C++(1980 頃 ~)
事象駆動型のシミュレーションを記述するために、C 言語を土台にして Simula67 のクラスの概念等を取り込む形で Bjarne Stroustrup(AT&T ベル研) によって 1979 年秋に設計が始まった言語である。当初は「クラス付きの C (C with classes)」と呼ばれていたが、1983 年夏に Rick Mascitti によって「C++」という名称が発案された。

知能情報システムプログラムにおける到達目標との対応:

対応	プログラムの到達目標
	（１）知識・理解
	a)
	b)
○	c) コンピュータのソフトウェアに関する基礎的知識を修得する。
	d)
	e)
	（２）当該分野固有の能力
	a)
	b)
○	c) プログラム等の要求条件を理解し、プログラム設計等の作業スケジュールを立て、プログラム作成等を計画通りに実行できる。
	（３）汎用的能力
	a)
	b)
	c)
	d)
	e)
	（４）態度・姿勢
	a)
	b)
	c)

0.3 教科書、参考書

教科書：

講義ノート等を pdf の形で Web に配置しておくので、各自で download を行い必要な箇所の印刷を行なって下さい。

C++言語に関する参考書：

- I. ポール「(Information&Computing 62) C プログラマのための C++ —オブジェクト指向プログラミングに向けて」(1992 年, サイエンス社, 絶版)
- I.Pohl「C++ for C programmers 3rd Edition」(1999 年, Addison-Wesley,$44.99)
- 柴田望洋「C プログラマのための C++入門」(1992 年, ソフトバンク, 絶版; 1999 年に「新装版 C プログラマのための C++入門」, ソフトバンク, 絶版)
- 塚越一雄「(Software Technology 25) 決定版 はじめての C++」(1999, 技術評論社,2680 円＋税)
- 柴田望洋「新版 明解 C++ 入門編」(2009 年, ソフトバンククリエイティブ,2700 円＋税)
- 柴田望洋「新版 明解 C++ 中級編」(2014 年, ソフトバンククリエイティブ,2700 円＋税)
- H. シルト「独習 C++ 第 4 版」(2012 年, 翔泳社, 3200 円＋税)
- B. ストラウストラップ「プログラミング言語 C++ 第 4 版」(2015 年, ソフトバンククリエイティブ,8800 円＋税)

- B. ストラウストラップ「C++によるプログラミングの原則と実践」(2016 年, ソフトバンククリエイティブ,7000 円＋税)
- B. ストラウストラップ「C++のエッセンス」(2015 年, ソフトバンククリエイティブ,2200 円＋税) ... ストラウストラップ (2015, 第 4 版) 第 I 部 (1〜5 章) とほぼ同じ
- S. メイヤーズ「Effective C++ 第 3 版」(2014 年, 丸善出版,3800 円＋税)
- 高橋麻奈「やさしい C++ 第 4 版」(2012 年, ソフトバンククリエイティブ,2600 円＋税)
- 粂井康孝「猫でもわかる C++プログラミング 第 2 版」(2015 年, ソフトバンククリエイティブ,2200 円＋税)

オブジェクト指向に関する参考書：

- 平澤章「オブジェクト指向でなぜつくるのか」(2004 年, 日経 BP 社,2400 円＋税)
- 山田隆太&(株) 豆蔵「豆蔵セミナーライブオンテキスト 1 わかるオブジェクト指向」(2005 年, 技術評論社,2480 円＋税)
- T.A. バッド「オブジェクト指向プログラミング入門 第 2 版」(2002 年, ピアソン, 4800 円＋税)
- I. グラハム「オブジェクト指向概論 第 2 版」(1996 年, トッパン,4660 円＋税)
- L.Cardelli&P.Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, Vol.17, No.4, pp.471–522, 1985.
- Yoo Hong Jun「図解 Java 流オブジェクト指向」(1998 年, 技術評論社,1880 円＋税)
- 日経ソフトウェア編「ゼロから学ぶ! 最新 Java プログラミング」(2009, 日経 BP 社, 2400 円＋税)
- 日経ソフトウェア編「Java ツール完全理解」(2011, 日経 BP 社, 2667 円＋税)
- 結城浩「(増補改訂版)Java 言語で学ぶデザインパターン入門」(2004 年, ソフトバンク, 3800 円＋税)
- 戸松豊和「増補改訂 Java プログラムデザイン」(1998 年, ソフトバンク, 2700 円＋税)

0.4 授業予定

	3 限	4 限
1 回	クラス定義以前の基本事項に関して、C 言語との違いを理解する。	**実習課題 1**: C 言語との違い、新しいプログラミング環境に慣れるための課題に取り組み、LaTeX でレポートを完成させる。(g++コンパイラ, ストリーム入出力, 等)
2 回		**実習課題 2**: 引き続き、C 言語との違い、新しいプログラミング環境に慣れるための課題に取り組み、LaTeX でレポートを完成させる。(暗黙の実引数, 関数の多重定義, 参照宣言, 空き領域演算子,string 型, 等)
3 回	クラスを定義してそのインスタンスを生成して使う、という考え方を理解する。	**実習課題 3**: クラスを定義してそのインスタンスを生成して使う、という方式に慣れるための課題に取り組み、LaTeX でレポートを完成させる。

4 回	何をクラスとして定義すべきか考える。 • 演算子の多重定義	**実習課題 4**: 定義するクラスの選択／設計を特に注意深く行なってもらうための課題に取り組み、LaTeX でレポートを完成させる。
5 回	• 継承 (既定義クラスの拡張), • 多態性の実現, 抽象クラス	
6 回	•Makefile を用いた分割コンパイル	**実習課題 5**: 継承, 多態性, 抽象クラスについての理解を深めるための課題に取り組み、LaTeX でレポートを完成させる。
7 回	• 型パラメータ付きのクラス定義, • ジェネリックプログラミング, • 標準テンプレートライブラリ (STL), • 例外処理, • オブジェクト指向のまとめ	
8 回	ターム末試験	**実習課題 6**: ジェネリックプログラミングや例外処理についての理解を深めるための課題に取り組み、LaTeX でレポートを完成させる。

1 C++プログラミング事始め

- C++言語の歴史, 特徴
- オブジェクト指向の考え方、利点
- 拡張子, g++コンパイラ
- C 標準ライブラリ関数の利用
- 簡単なプログラム例

1.1 C++言語の歴史

{Stroustrup(2015, 言語 C++第 4 版)1.4 節 }

1979 年秋　Bjarne Stroustrup(AT&T ベル研) が「クラス付きの C (C with classes)」言語の設計作業を開始。

(目的： 事象駆動型のシミュレーションを行い、マルチプロセッサと LAN のための UNIX カーネルサービスの分散を行えるようにする。
設計方針：C 言語 (ハードウェアを直接的に処理可) に Simula(シミュレーションを記述可) スタイルのクラス機能を加え、更に型チェック等を改善。
最初の機能：クラス、派生クラス、public/private によるアクセス制御、コンストラクタとデストラクタ、引数型チェックと暗黙の型変換機能をもつ関数宣言、ノンプリエンプティブな並行タスクを扱うためのライブラリ、乱数生成をサポートするためのライブラリ、.....)

1983 年夏　Rick Mascitti が「C++」という名前を発案。

(追加機能：仮想関数、関数と演算子の多重定義、参照、ストリーム I/O、複素数ライブラリ、.....)

1984 年　名前を「クラス付きの C (C with classes)」から「C++」に変更。

1985 年 10 月　C++言語の最初の商用リリース。

1986 年　B.Stroustrup「The C++ Programming Language」(Addison-Wesley)

1990 年　M.A.Ellis&B.Stroustrup「The Annotated C++ Reference Manual」(通称 ARM C++; Addison-Wesley)

1991 年　B.Stroustrup「The C++ Programming Language, 2nd edition」(Addison-Wesley)

(追加機能：総称型、例外処理の機構、.....)

1997 年　B.Stroustrup「The C++ Programming Language, 3rd edition」(Addison-Wesley)

(追加機能：名前空間、`dynamic_cast`、テンプレートに関する数多くの改良、汎用的なコンテナとアルゴリズムによる STL フレームワーク、.....)

1998 年　ISO による C++ の標準規格。(通称 C++98)

2011 年　ISO による C++ の新しい標準規格。(通称 C++11)

1.2 C++言語の特徴

{Pohl(1999)p.4 の 2-4 行目, p.18summary,
Stroustrup(2015, 第 4 版)1.2 節,1.3.3 節}

- 1979 年頃の C 言語と Simula67 言語のクラスの概念を土台にしている。

  クラス定義の機構を用いて問題領域に固有の**抽象データ型** を実現できる。
  $\Longrightarrow$ 問題領域を適切にモデル化できる。
  $\Longrightarrow$ 分り易いプログラム
  例外処理の機構がある。 $\Longrightarrow$ 本来の処理の流れの明瞭さを損なわずに、エラー処理をきちんと行える。
  .....

- 汎用プログラミング言語だが、(C 言語と同様に) システムプログラミング用言語の性格も強い。(Stroustrup,2015, 第 4 版,1.2 節)

- C 言語のプログラムはほぼそのまま C++のプログラムとしても働く。

  C 言語プログラマへの提言 (Stroustrup,2015, 言語 C++第 4 版,1.3.3 節):
  C++を幾つかの機能が追加された C 言語と考えない様に。C++の C 言語より優位な点の恩恵を受けるには、C 言語の場合とは異なる設計/ 実装のスタイルが必要だ。

- **オブジェクト指向**プログラミングをサポートする。
  $\Longrightarrow$ 既定義クラスの拡張 (継承) を行うことにより、コードの再利用が促進される。

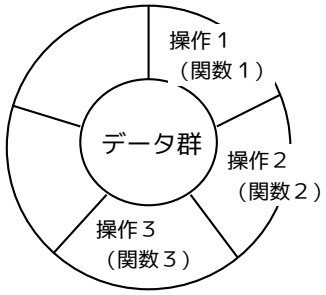
- **STL**(Standard Template Library) が用意され、多くの標準的なデータ構造や関連アルゴリズムを利用できる。(利用の際は、パラメータ付きデータ型を用いた**総称的プログラミング**を行うことになる。)

1.3 オブジェクト指向の考え方、利点

{Pohl(1999)1.1 節,1.9 節 }

オブジェクト指向の考え方:

- 関連するデータ群と操作 (関数) を 1 つにまとめて**カプセル化**し、ソフトウェア部品として扱うことが基本。(この種の部品は C 言語における構造体を拡張したもので、これらを**オブジェクト**と呼ぶ。)



- C++プログラム上では、構造体の場合と同じ様に、まず
  ① オブジェクトの構成要素等を記述した「**クラス**」を定義してオブジェクトの型枠／設計図を作り、
  ② その型枠／クラス定義に沿ったオブジェクトを必要なだけ用意して使う、
  という作業を行うことになる。
- オブジェクト内のデータについては、外部からのアクセスを制限 (i.e. **データ隠蔽**) する。
  $\Longrightarrow$ オブジェクト毎の独立性／**モジュール性**が高くなり、**作成・保守が容易**になる。
  $\Longrightarrow$ アクセス制限されていて隠蔽されている内部の実装方法 (e.g. データ構造) については、他のオブジェクトとは独立に自由に改良を進めることができる。
- プログラミング言語内に、既定義クラスを拡張するための機構を用意する。
  $\Longrightarrow$ 既定義クラスの拡張 (**継承**) を行うことにより、**コードの再利用**が促進される。

- 既定義クラスの拡張を繰り返すと、クラス間の階層構造ができる。そこで、プログラミング言語内に、この階層構造を利用して、類似したオブジェクトを統合的に扱うための機構 (i.e. **多態性**をもったコードを可能にする機構) を用意する。

オブジェクト指向の難点、利点 (まとめ)：

- **(難点)**「適切なクラスを設計する」stepも必要になるので、C言語等を用いた通常の手続き型プログラミングと比べて、**プログラミング作業は複雑**になる。

しかし、クラスの設計／定義を適切に行えば、

- **(利点)** ソフトウェア部品の独立性／モジュール性が高くなり、**保守を行い易く**なる。
- **(利点) コードの再利用**が促進される。
- **(利点) 類似したオブジェクトの統合的な扱い**が可能になる。

⇨ 大きなソフトウェアを構築する時は、オブジェクト指向は特に有効。

1.4 Linuxの下でのC++プログラミング作業

{塚越 (1999)p.25,p.29, Pohl(1999)p.31,p.29, 柴田 (2009 入門編)p.5, 柴田 (1992)p.18, ウィキペディア, 粂井 (2015)p.19}

拡張子： C言語ソースファイルの拡張子としては `.c` だけが用いられているが、C++言語ソースファイルの拡張子としては次のようなものが用いられている。

`.C` `.cc` `.cxx` `.cpp` `.c`

⇨ この授業では `.cpp` を拡張子として用いる。

C++言語のコンパイラ： C++言語の処理系としてはVisual Studio Express (Windows用) 等の統合開発環境もあるが、この授業ではLinux仮想端末のコマンドライン上で動作するGNU C++ コンパイラを用いる。コマンド名は `g++` で、使い方は `gcc` コマンドとほぼ同じである。例えば、コマンドライン上で

`g++` 半角空白で区切られた `C++`ソースファイル や `.o` ファイル の列

とすると、カレントディレクトリ上に `a.out` という名前の実行ファイルができる。

補足 (g++と gcc の関係)：

- 柴田 (1992)p.18 や Pohl(1999)p.31 の記述によると、昔は C++コンパイラ (e.g. g++コマンド) は
  ① C++のソースを C のソースに変換し、
  ② 変換結果を C コンパイラでコンパイルする、
  という作業を行うトランスレータとして実現されていた。
- 塚越 (1999)p.29 の記述によると、昔は g++コマンドは
  C++のソースをコンパイルするための引数を付けて gcc を呼び出す、
  という形で実装されていたこともあったみたい。

(続く)

補足 (g++と gcc の関係, 続き)：
- 仮想端末上で man gcc と打ち込んで現れる電子マニュアルによると、
  - ◇ 名称 gcc, g++ — GNU プロジェクト C および C++ コンパイラ
  - ◇ C と C++ のコンパイラは統合されています。ソースファイル名の拡張子によってソースの言語を識別しますが、デフォルトの動作は、どちらの名前でコンパイラを使うかに依存しています。
    gcc ... プリプロセス済みの (.i) ファイルを C のファイルと仮定し、C スタイルのリンクを行います。
    g++ ... プリプロセス済みの (.i) ファイルを C++のファイルと仮定し、C++ スタイルのリンクを行います。
  - ◇ ソースファイル名の拡張子は、その言語が何であるかと、どのような処理が行われるべきかを示します。
    .c ... C 言語ソースです。
    .C ... C++言語ソースです。
    .cc ... C++言語ソースです。
    .cxx ... C++言語ソースです。
    .m ... Objective-C 言語ソースです。
    .o ... オブジェクトファイルです。
    .a ... アーカイブファイルです。
  - ◇ -x c++ というオプション指定によって、続く入力ファイルの言語を c++ と明示することができる。
- ウィキペディア (表題「GNU コンパイラコレクション」) によると、
  - ◇ GNU Compiler Collection は、GNU のコンパイラ群である。略称は「GCC」。
  - ◇ 標準パッケージには C、C++、Objective-C、Objective-C++、Fortran、Java、Ada、Go のコンパイラ並びにこれらのライブラリが含まれている。
  - ◇ 当初は C コンパイラとして開発し、GCC は GNU C Compiler を意味していた。しかし、もともと多言語を想定して設計しており、GNU C Compiler と呼ばれていたときでも多くの言語に対応していた。現在でも GNU C Compiler の意味で「GCC」と呼ぶことも多い。ちなみに GNU C Compiler の実行ファイルの名称も gcc である。なお、GNU C++コンパイラを G++、GNU Java コンパイラを GCJ、GNU Ada コンパイラを GNAT と呼ぶ。

  要するに、GCC は 昔はC 言語のコンパイラだったが、今は 多言語を対象にしたコンパイラ群を意味する、ということ。

C 言語標準ライブラリ関数の利用： C++プログラム内では、C 言語の標準ライブラリ関数を必要に応じて使うことができる。但し、その際の #include 指令は次の様に書く。

```
#include <c対応するヘッダファイルの拡張子を除いた名前>
```

例えば、C 言語の scanf() や printf() を用いたい場合は、C++プログラムの前の方に次の#include 指令を置く。

```
#include <cstdio>
```

1.5 簡単なプログラム例

{Pohl(1999)pp.4-5,pp.7-8, Pohl(1992)pp.73-74, Stroustrup(2015, 第 4 版)7.3.2 節, 柴田 (1992)p.18}

ここでは、C++プログラムの雰囲気に慣れるために簡単な C++プログラムの例を幾つか示す。

例 1.1 (決められた文字列の出力) 会話画面に「Welcome to C++ World!」と表示するだけの次の C プログラムを考える。

```
[motoki@x205a]$ cat -n printWelcomeToCppWorld.c
```

```
 1 /* 「Welcome to C++ World!」と出力するCプログラム */
 2 #include <stdio.h>
 3
 4 void printMessage(char *message);
 5
 6 int main(void)
 7 {
 8   printMessage("Welcome to C++ World!");
 9   return 0;
10 }
11
12 /* 引数で与えられた文字列を出力 */
13 void printMessage(char *message)
14 {
15   printf("%s\n", message);
16 }
[motoki@x205a]$ gcc printWelcomeToCppWorld.c
[motoki@x205a]$ ./a.out
Welcome to C++ World!
[motoki@x205a]$
```

このプログラムは、次の様に少しだけ手直しすれば、C++プログラムとして動作する。

```
[motoki@x205a]$ cat -n printWelcomeToCppWorld2.cpp
 1 /* 「Welcome to C++ World!」と出力するC++プログラム */
 2 #include <cstdio>
 3
 4 void printMessage(const char *message);
 5
 6 int main(void)
 7 {
 8   printMessage("Welcome to C++ World!");
 9   return 0;
10 }
11
12 /* 引数で与えられた文字列を出力 */
13 void printMessage(const char *message)
14 {
15   printf("%s\n", message);
16 }
[motoki@x205a]$ g++ printWelcomeToCppWorld2.cpp
[motoki@x205a]$ ./a.out
Welcome to C++ World!
[motoki@x205a]$
```

ここで、

- プログラム 2 行目 の#include 指令は、C 言語の printf() 関数を正しくコンパイルするために置いてある。(これによって、C 言語用に用意されたヘッダファイル /usr/include/stdio.h の中身をこの場所に挿入する作業を前処理として行うことをコンパイラに指示している。)
- プログラム 4 行目,13 行目 の const **修飾子**は、引数結合で受け取った文字列を関数本体内で書き換えないことを宣言している。 (**補足**：実際にプログラム 4 行目,13 行目に const 修飾子を付けないと、コンパイル時に「警告: deprecated conversion from string constant to "char*"」という警告が出る。これは、const 宣言がなく、単に「char *message="Welcome to C++ World!"」という引数結合を行っただけだと、呼び出された関数本体内で例えば「message[15]='w';」という、**文字列リテラル** (i.e. 定数文字列) 領域への書き込みを行なってしまう危険性があるためである。C 言語ではプログラマの責任という考えで何の警告も出なかったが、C++言語においては この様に安全でない箇所に関してはコンパイラがエラーと見做して処理を行う。)

例 1.2 (ストリーム I/O, 名前空間) 1 つ前の例 1.1 で示した C++プログラム printWelcomeToCppWorld2.cpp と同等の働きをする、もっと C++言語の書き方に沿った C++プログラムを次に示す。

```
[motoki@x205a]$ cat -n printWelcomeToCppWorld3.cpp
     1 /* 「Welcome to C++ World!」と出力する C++プログラム */
     2 #include <iostream>  // ストリーム I/O を使うので
     3 using namespace std; // 名前を特定する際のデフォルトの名前空間を
     4                      //  std(標準ライブラリ内の名前を含む) と指定
     5 void printMessage(const char* message);
     6
     7 int main()
     8 {
     9   printMessage("Welcome to C++ World!");
    10 }
    11
    12 /* 引数で与えられた文字列を出力 */
    13 void printMessage(const char* message)
    14 {
    15   cout << message << endl;
    16 }
[motoki@x205a]$ g++ printWelcomeToCppWorld3.cpp
[motoki@x205a]$ ./a.out
Welcome to C++ World!
[motoki@x205a]$
```

ここで、

- プログラム 2~4 行目 の//は、これ以降の行末までの部分がコメントであることを表す。
- プログラム 5 行目,13 行目 では「char *message」ではなく「char* message」と書いている。文法的には同じことであるが、C++言語では*(や&) を型名の方に寄せて、データ型を表す部分をより明確に示す習慣がある。

- C++言語では、関数引数部に何も指定しないと引数なしと認識されるので、プログラム7行目では main(void) とせずに単に main() と書いた。(C言語では、関数引数部に何も指定しないと「引数チェックを行わない」と認識される。)
- C++言語では、main() 関数の暗黙の戻り値として 0 が仮定されているので、プログラム9~10行目の間に return 0; という行を置くのを省いている。(C言語では、main() 関数の戻り値を指定するのを省くと、コンパイル時に警告が出る。)
- C++プログラムでは、通常、C言語用に用意された scanf(),printf() 等の標準ライブラリは使わずに、代わりに**ストリームI/O**と呼ばれる入出力システムが用いられる。その使用例がプログラム15行目に見られる。この行では、
  - ◇ cout は、**標準出力ストリーム**を表すオブジェクト(ソフトウェア部品)の名前である。ヘッダファイル iostream.h の中で定義されているので、2行目で#include の指示をしている。
  - ◇ << は**左結合性**をもった(i.e.「a<<b<<c」を「(a<<b)<<c」の略記と考える)二項演算子で、プログラム15行目の場合は次の様に働く。
    - ①cout << message ... (副作用として)第2オペランドとして指定された文字列 message を標準出力ストリームに流し込み、演算結果として第1オペランドのオブジェクト(cout)を返す。(演算結果として返される cout は、内部の出力バッファに先ほど流し込まれた文字列 message が記録されており、第1オペランドとして与えられた時と内部状態が変わっている。)
    - ②cout << endl ... (第2オペランドとして endl が指定されている場合は、副作用として)標準出力ストリーム cout 内部の出力バッファに溜まったデータと改行コードを順に標準出力に吐き出し、演算結果として第1オペランドのオブジェクト(cout)を返す。
  - ◇ endl は、**マニピュレータ**(manipulator,**操作子**)と呼ばれるものの一種である。マニピュレータの実体は関数へのポインタで、「cout << endl」の場合は endl(cout) という処理が行われることになる。
- 複数の所で作られたプログラムを合わせて使おうとした時、(関数名,構造体名,クラス名,... 等の)大域的な名前が衝突することがある。こういった不都合な事態を避けるために、局所的でない個々の宣言を特定の**名前空間**に所属させ、非局所的な名前を

  その名前の宣言された名前空間の名前 :: その名前

  という風に明示的に表す方式が導入された。プログラム3行目の using 指令は、標準ライブラリの名前空間 std を暗黙の名前空間として扱うことを宣言している。この using 指令がない場合は、プログラム15行目の cout の部分は名前空間を明示して std::cout と書く必要がある。

例 1.3 (関数の暗黙の実引数, inline 修飾子) 1つ前の例1.2で示したC++プログラム printWelcomeToCppWorld3.cpp については、関数 printMessage() の機能を次の様に少しだけ拡張することができる。

```
[motoki@x205a]$ cat -n printWelcomeToCppWorld4.cpp
     1 /* 「Welcome to C++ World!」と出力するC++プログラム */
     2 #include <iostream>  // ストリームI/Oを使うので
     3 using namespace std;
```

```
 4
 5 inline void printMessage(const char* message="ProgrammingAII");
 6
 7 int main()
 8 {
 9   printMessage("Welcome to C++ World!");
10   printMessage();
11 }
12
13 /* 引数で与えられた文字列を出力 */
14 inline void printMessage(const char* message)
15 {
16   cout << message << endl;
17 }
[motoki@x205a]$ g++ printWelcomeToCppWorld4.cpp
[motoki@x205a]$ ./a.out
Welcome to C++ World!
ProgrammingAII
[motoki@x205a]$
```

ここで、

- プログラム5行目の="ProgrammingAII"は、関数printMessage()が実引数を省略して呼び出された時に実引数として "ProgrammingAII" が**暗黙に**用いられることを指示している。(**補足**：この種の指示は、実際に実引数を省略する関数呼び出しを行う前に配置しないと、コンパイル時に「エラー: too few arguments to function "void printMessage(std::string)"」というメッセージが出る。また、5行目だけでなく14行目の関数頭部にもこの指示を入れると、「エラー: default argument given for parameter 1 of "void printMessage(std::string)" [-fpermissive]」というメッセージが出る。)
- プログラム5行目,14行目の inline **修飾子**は、**可能であれば**、引数付きマクロと同じ様に、関数呼び出し場所に関数本体のコードを埋め込むことをコンパイラに指示している。(この種の指示は、引数付きマクロと違って無視されることもある。)

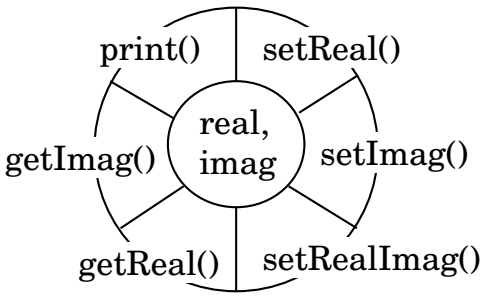
例 1.4 (操作方法も要素に含む構造体) C言語では関連するデータを1つにまとめて構造体として表すことができた。C++言語では、この構造体の中に「データの操作方法」である関数を含ませることができる。この具体例として、複素数の実部データ real, 虚部データ imag, データ操作のための関数群 setReal(), setImag(), setRealImag(), getReal(), getImag(), print() を構成要素とする構造体を定義し、使用テストを行うC++プログラム testComplexNumberStruct1.cpp を次に示す。

```
[motoki@x205a]$ cat -n testComplexNumberStruct1.cpp
     1 /* 操作方法も要素に含む構造体 ComplexNumber を定義しテストする
                                                C++プログラム */
     2 #include <iostream>  // ストリーム I/O を使うので
```

```
 3 using namespace std;
 4
 5 struct ComplexNumber {
 6   double real;
 7   double imag;
 8   void setReal(double real){ this->real = real; }
 9   void setImag(double imag){ this->imag = imag; }
10   void setRealImag(double real, double imag){
11     this->real = real;
12     this->imag = imag;
13   }
14   double getReal(){ return real; }
15   double getImag(){ return imag; }
16   void print() const {
17     cout << "(" << real << ")+(" << imag << ")i" << endl;
18   }
19 };
20
21 int main()
22 {
23   ComplexNumber z;
24
25   z.setRealImag(9.5, -3);
26   z.print();
27 }
[motoki@x205a]$ g++ testComplexNumberStruct1.cpp
[motoki@x205a]$ ./a.out
(9.5)+(-3)i
[motoki@x205a]$
```

ここで、

- プログラム8〜18行目に構造体内部のデータreal, imagに関する操作方法が関数として定義されている。この内、setReal(), setImag(), setRealImag()の3つは内部データへの値の設定のための操作方法を提供している。また、getReal(), getImag()の2つは内部データの値を閲覧するための操作方法を提供し、print()は内部データの表す複素数を出力するための操作方法を提供している。



ここに挙げたソースプログラム中では4つの関数setReal(), setImag(), getReal(),

getImag() は使われていないが、定義する構造体に汎用性を持たせるために構造体の要素として含ませている。

- プログラム 8~9 行目,11~12 行目 の this は自己構造体 (オブジェクト) へのポインタを表す。従って、8 行目 の this->real は構造体 (オブジェクト) 自身が内部に保有しているデータ領域 real を表し、8 行目 代入演算子=の右側に現れる real は関数 setReal() の仮引数を表す。
- プログラム 14~15 行目 に現れる real，imag はそれぞれ構造体 (オブジェクト) 自身が内部に保有しているデータ領域 real，imag を表す。
- プログラム 16 行目 の const 宣言は、関数本体内で構造体内部に保有しているデータ領域の値を変更しないことを宣言している。
- プログラム 23 行目 の ComplexNumber は、上で定義した構造体のデータ型を表す。C 言語では「struct ComplexNumber」と書く必要があったが、C++言語では 構造体タグの名前をデータ型名として使うことができる。
- プログラム 25~26 行目 では、それぞれ構造体 z 内に含まれる関数 setRealImag()，print() を呼び出している。

例 1.5 (アクセス指定子 private，public) 1 つ前の例 1.4 で示した C++プログラム testComplexNumberStruct1.cpp については、構造体 ComplexNumber 内の要素への外部からのアクセスを次の様に制限しデータ隠蔽を進めることができる。

```
[motoki@x205a]$ cat -n testComplexNumberStruct2.cpp
     1 /* 操作方法も要素に含む構造体 ComplexNumber を定義しテストする
                                                    C++プログラム */
     2 #include <iostream>  // ストリーム I/O を使うので
     3 using namespace std;
     4
     5 struct ComplexNumber {
     6 private:
     7   double real;
     8   double imag;
     9 public:
    10   void setReal(double real){ this->real = real; }
    11   void setImag(double imag){ this->imag = imag; }
    12   void setRealImag(double real, double imag){
    13     this->real = real;
    14     this->imag = imag;
    15   }
    16   double getReal(){ return real; }
    17   double getImag(){ return imag; }
    18   void print() const {
    19     cout << "(" << real << ")+(" << imag << ")i" << endl;
    20   }
    21 };
```

```
    22
    23 int main()
    24 {
    25   ComplexNumber z;
    26
    27   z.setRealImag(9.5, -3);
    28   z.print();
    29 }
[motoki@x205a]$ g++ testComplexNumberStruct2.cpp
[motoki@x205a]$ ./a.out
(9.5)+(-3)i
[motoki@x205a]$
```

ここで、

- プログラム 6 行目 の private 指定は、この行以降 (別の指示があるまで) の構造体要素への外部からのアクセスを禁止することを宣言している。
- プログラム 9 行目 の public 指定は、この行以降 (別の指示があるまで) の構造体要素への外部からのアクセスを許可することを宣言している。

例 1.6 (クラス定義) 1 つ前の例 1.5 で示した C++プログラム testComplexNumberStruct2.cpp の中で、キーワード struct を用いて定義されたオブジェクトの設計図／型枠は、キーワード class を用いて次の様に定義することもできる。 (機能的に同じものであるが、キーワード class を用いて定義される設計図／型枠のことを「構造体 (の枠組み)」ではなくクラスと呼んでいる。)

```
[motoki@x205a]$ cat -n testComplexNumberClass.cpp
     1 /* クラス ComplexNumber を定義しテストする C++プログラム */
     2 #include <iostream>  // ストリーム I/O を使うので
     3 using namespace std;
     4
     5 class ComplexNumber {
     6   double real;
     7   double imag;
     8 public:
     9   void setReal(double real){ this->real = real; }
    10   void setImag(double imag){ this->imag = imag; }
    11   void setRealImag(double real, double imag){
    12     this->real = real;
    13     this->imag = imag;
    14   }
    15   double getReal(){ return real; }
    16   double getImag(){ return imag; }
    17   void print() const {
    18     cout << "(" << real << ")+(" << imag << ")i" << endl;
```

```
19   }
20 };
21
22 int main()
23 {
24   ComplexNumber z;
25
26   z.setRealImag(9.5, -3);
27   z.print();
28 }
[motoki@x205a]$ g++ testComplexNumberClass.cpp
[motoki@x205a]$ ./a.out
(9.5)+(-3)i
[motoki@x205a]$
```

ここで、

- プログラム5行目先頭のキーワードがstructからclassに変わっていることに注意して下さい。
- プログラム5~6行目の間にprivate指定が無いのは、キーワードclassを使った場合には要素へのデフォルトのアクセス指定がprivateであるためである。(これに対し、キーワードstructを使った場合には要素へのデフォルトのアクセス指定がpublicとなる。2つのキーワードstruct，classの使い方の違いはこれだけである。)

2 「オブジェクト指向」以外でのC言語の拡張箇所

- コメント
- 識別子とキーワード
- 基本データ型
- リテラル
- 変数の宣言,for 文制御変数の局所的な宣言,const 修飾子
- 型変換
- スコープ解決演算子
- ストリーム I/O
- 演算子
- 制御構造
- main() の暗黙の戻り値, 関数引数リストが空の場合, 関数の暗黙の実引数, 関数の多重定義, 関数の inline 宣言
- Namespace
- 参照宣言
- 空き領域演算子 new と delete
- 抽象データ型 string と vector<>

2.1 コメント

{Pohl(1999)2.1.1 節 }

- /* から */ までは、C 言語の場合と同様にコメント (comment, 人間向けの注釈)。
- // から その行の最後まではコメント。

補足 (**TODO** コメント):
既存のプログラムを眺めていると、

```
//TODO ...
```

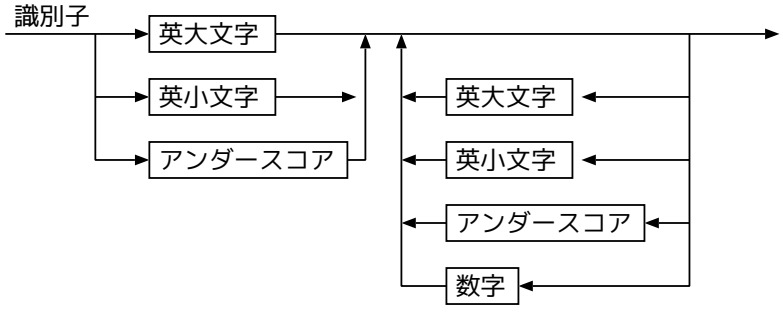
という風に “TODO” や “FIXME”, “XXX” で始まるコメントに出会うことがある。これらの意味は次の通りである。

- `//TODO` 作業 ··· 作業 ということをやらなければならない。(to do)
- `//FIXME` 修正作業 ··· 正しく動いてないので 修正作業 に記述されている様な修正が必要。(fix me)
- `//XXX` 間違い ··· とりあえず動いてはいるが 間違い に記述されている様に正しくない。

2.2 識別子とキーワード

{Pohl(1999)2.1.2–3 節, 柴田 (2009)p.70}

識別子： 変数等に付ける名前としては、次のものが許される。



但し、キーワード(下記)は識別子として使うことはできない。また、連続したアンダースコア(下線, _)を含む文字列、および1文字目がアンダースコアで2文字目が英大文字の文字列は言語処理系内で既に使われている可能性があるので、識別子としての使用は避けるべきである。

キーワード： 次の通り。

asm	auto	bool	break	case	catch
char	class	const	const_cast	continue	default
delete	do	double	dynamic_cast	else	enum
explicit	export	extern	false	float	for
friend	goto	if	inline	int	long
mutable	namespace	new	operator	private	protected
public	register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch	templat
this	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while			

コーディング規約： 一般に、コードの可読性・保守性を高めたり不具合を招く可能性を抑制したりするために、処理手順をプログラムとしてコード化する際に守るべき**コーディング規約**が設定されていることがある。例えばJava言語の場合は次の様なものがある。

(日経ソフトウェア編「ゼロから学ぶ！最新 Java プログラミング」p.156 表 1, 一部手直し)

名称	Web ページのアドレス
電通国際情報サービス版 Java コーディング規約 2004	http://www.objectclub.jp/community/codingstandard/JavaCodingStandard2004.pdf
オブジェクト倶楽部版 Java コーディング規約	http://www.objectclub.jp/community/codingstandard/CodingStd.pdf
Java 言語コーディング規約 (**Sun コーディング規約**)	http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html (原文), http://www.oracle.com/technetwork/java/codeconvtoc-136057.html (原文), http://numata.designed.jp/javacodeconv/ (和訳)
頑健な Java プログラムの書き方	http://www.alles.or.jp/~torutk/oojava/codingStandard/writingrobustjavacode.html

これらのコーディング規約の中には、変数名に対する命名の仕方、書式、不具合を招くコードと回避例、等が示されている。実際には、各々の規約は歴史的背景や規約策定者の好みに依存している部分も含まれているので、全てのコーディング規約が1つの方向に向かっている訳ではなく、互いに矛盾した指針もある。しかし、プログラムを書く際はこういった規約にも十分留意すべきである。例えば、使い捨ての一時的な変数の名前を除いて、識別子に対する名前の付け方として次の様な指針が一般的である。

◇ 英単語など、意味のある単語を並べて名前を付ける。

◇ 「意味のある単語」としては、不可解な短縮形は使わず、可能な限り省略なしの単語を用いる。(それによって、読み易く理解し易いコードになることが期待される。)

2.3 基本データ型

{Pohl(1999)2.4 節, 柴田 (2009)p.144}

- C 言語の基本データ型に加えて、bool と wchar_t が基本データ型となっている。
- bool 型は論理値 (i.e.{ 真, 偽 }) を値域とするデータ型である。bool 型の定数値を表すキーワードとして true(真) と false(偽) の 2 つが用意されている。(C 言語では int 型で代用されていた。)
- wchar_t 型は、255 文字を超える文字セット (e.g.Unicode) 内の文字を保持するためのデータ型である。
- bool 型も wchar_t 型も整数型の一種として分類される。実際、bool 型定数値 false の整数としての値は 0 であり、true の整数としての値は 1 である。また、(C 言語の場合と同様に) 0 以外の整数値は true(真) と見做される。

2.4 リテラル (定数)

{ 柴田 (2009),Stroustrup(2015, 第 4 版)6.2.3.2 節 }

C++では定数のことを**リテラル**と呼んでいる。

整数型リテラル：

- 10 進, 16 進, 8 進の表記が可能。(C 言語と同様。)
- int で表せれば int 型, さもなければ long 型。但し、最後に L または l を付けると long 型になる。

浮動小数点数型リテラル：

- C 言語の場合と同じ。(標準は double 型で、..... 。)

文字リテラル：

- C 言語の場合と同じで、'A' の様に文字を 1 重引用符で囲んで表す。
- 文字リテラルのデータ型は char。(C 言語では int 型だった。)
- コンピュータ上で採用されている文字符号体系に従って、各文字には文字コード (長さ 8 のビット列) が割り当てられている。C 言語の場合と同様、文字リテラルの値はこの文字コードの表す整数値 (文字番号) である。(C 言語の場合と同様、char は整数型の一種。)

論理型リテラル：

- true と false の 2 つ。

2.5 変数の宣言

{Pohl(1999)2.4.1 節,2.8 節,p.48,p.37,p.77, 柴田 (1992)p.11}

- C 言語と同様の書き方。
- 局所変数の場合は、C 言語の場合と同様に、宣言によって (暗黙値に) 初期化される訳ではない。
- C++言語の場合は、変数の宣言も**文** (statement) の一種と考え、プログラムのどこに変数宣言を置いてもよい。(但し、使う前に宣言しておかなければならない。C 言語では、変数宣言は文の扱いではなく、ブロックの最初に固めて置くことになっていた。)
- プログラムの途中で変数宣言を行う場合は、(変数の) 名前の有効範囲に気を付けなければならない。
- C++言語の場合は、`for` 文中のループ制御変数の値を初期設定する場所で、ループ制御変数の宣言を行うこともできる。例えば、

```
for (int i=0; i<n; i++) {
    
}
```

と書くことができ、この場合、ループ制御変数 `i` はこのループの中だけで有効になる。

`const` 修飾子：

- 値の初期設定を伴う変数宣言の前に `const` というキーワードを置くと、その変数は値の変更を行えなくなる。(その変数への代入を行う文を書くとコンパイルエラーとなる。)
- `const` 宣言された変数は**定数式** (i.e. 定数だけから構成される式) の一部として使うことができ、それゆえ配列宣言の際の要素数指定に使うことができる。(C 言語では、定数式の中に変数を使うことは出来ない。)
- **(引数なし) マクロの代わりに `const` 宣言された変数を用いることができる。**
- 関数の外で `const` 宣言された変数の名前は、(デフォルトでは) そのソースファイル内だけで有効で、他のソースファイルからも見える様にするためには、`const` 修飾子の前に明示的にキーワード `extern` を置く必要がある。

(参考) クラス, 局所変数, 定数変数に対する命名規則： Java 言語における Sun コーディング規約では、識別子名の付け方に関する一般的な指針 (2.2 節) に加えて、クラス, 変数に対する名前の付け方として次の様な指針を設けている。

- **クラス名, 変数名の付け方 (定数値を保持する領域は除く)：**
  - ◇ 主として英小文字を用いる。
  - ◇ 2 つ以上の「意味のある単語」から構成する場合は、(単語の区切を明らかにするために)2 つ目以降の単語の頭文字を大文字にする。
  - ◇ クラス名の場合 は、先頭文字を大文字にし、名詞 (句) を表す単語列にする。
  - ◇ 変数名の場合 は、先頭文字を小文字にする。
- **定数値を保持する変数名の付け方：**
  - ◇ 「意味のある単語」を構成する全ての英文字を大文字にする。
  - ◇ 2 つ以上の「意味のある単語」から構成する場合は、(単語の区切を明らかにするために) 単語間にアンダースコア (_, 下線記号) を入れる。

2.6 型変換

{Pohl(1999)2.5 節, Stroustrup(2015, 第 4 版)11.5.2 節 }

算術計算の際の自動型変換： C 言語の場合と同様に、算術計算の際は、内部では次の順に強制的に型変換が行われる。

① `bool` 型や `char` 型,`short` 型, 列挙型のデータは演算前に `int` 型に変換される。また、`int` 型で表せない値は `unsigned` 型に変換される。

② 異なる型のデータ間で演算する場合、型を揃えるために、データ型間の次の順序に従って、演算前に下位 (i.e. 左) の方の型が上位の型に変換される。(変換後の型が演算結果の型になる。)

`int` < `unsigned` < `long` < `unsigned long`
< `float` < `double` < `long double`

$\Longrightarrow$ `char` 型同士の加算結果は `char` ではなく `int` 型。

キャスト (型変換)： 式の値をデータ型という型に変換したい時、明示的な型変換のために、C 言語, 古い C++言語に従った次の書き方ができる。

- C 言語の場合と同様に、
  (データ型) 式
- 関数記法 (古い C++):
  データ型 (式)

ただ、古いキャスト記法では、安全で何の問題もない型変換も、的確でなくコンパイラによる正当性チェックも難しい型変換も、1 種類の演算子／関数に押し込められていて、過度の利用がエラーの大きな原因となることがある、ということが指摘されていた。そこで、型変換を目に付き易くし、プログラマがキャストの意図を表現できるようにするために、次の**名前付きキャスト**が導入された。

- **安全**な (i.e. 暗黙の型変換も可能な様な) 型変換の場合
  `static_cast<`データ型`>(`式`)`
- 整数 $\leftrightarrow$ ポインタ間の様に変換結果がシステムに依存し安全でない場合 (出来れば使用を避ける)
  `reinterpret_cast<`データ型`>(`式`)`
- `const` 性,`volatile` 性 (揮発性) の属性を付けたり外したりしたい場合
  `const_cast<`データ型`>(`式`)`
- クラス階層で上位クラス (基底クラス) のオブジェクトを下位クラス (派生クラス) のオブジェクトと見做したい場合
  `dynamic_cast<`データ型`>(`式`)`

2.7 スコープ解決演算子

{Stroustrup(2015, 第 4 版)6.3.4 節, 柴田 (2009)6.3 節 }

スコープ： プログラム内に現れる名前は、それぞれに固有の有効範囲 (**スコープ**, scope, という) をもつ。スコープとしては、プログラム内の任意の範囲が可能な訳ではなく、次の範囲設定だけが可能である。

- **局所スコープ**... ブロック (i.e. 波括弧 { } で囲まれた範囲) 中の、局所名の宣言以降の部分。
- **クラススコープ**... クラスや構造体の定義を記述した範囲 (i.e. クラス定義開始直後の { とその対になっている } で囲まれた範囲)。
- **名前空間スコープ**... 名前空間の定義中で、所属する名前の宣言以降の部分。
- **大域スコープ (ファイルスコープ)** ... ソースファイル中で、関数定義やクラス定義、列挙クラス定義、名前空間定義の外に置かれた、大域的な名前の宣言以降の部分。

. . . . . .

スコープを指定した名前の参照： **スコープ解決演算子 (対象特定演算子)** : : を用いることにより、スコープを指定した名前の参照が可能である。例えば、

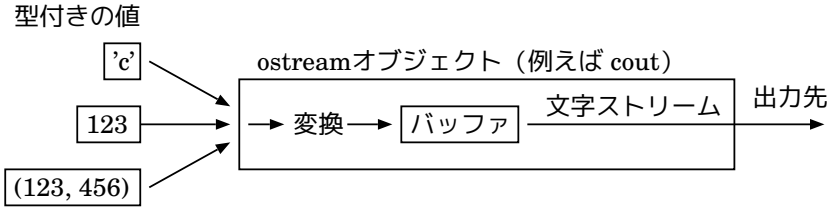
- : : 名前 ... ソースファイル (最大のファイルスコープ) 中の大域的な 名前
- 名前空間名 : : 名前 ... 名前空間 名前空間名 中に登録された関数, クラス, オブジェクト, 等の 名前
- クラス名 : : 名前 ... クラス クラス名 のスコープ中の要素の 名前

2.8 ストリーム入出力

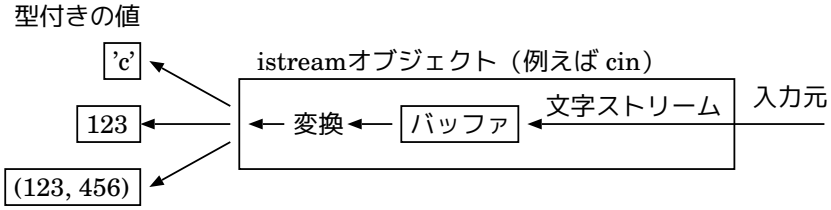
{Pohl(1999)2.2 節,D.1-4 節,D.8-9 節, 塚越 (1999)7.5 節,11.1-4 節, 柴田 (2009)3.6 節, Stroustrup(2015, 第 4 版)38.1 節,38.3-4 節}

C 言語と同じく、入出力機能は言語の一部として実装されている訳ではない。しかし、C 言語と違い「オブジェクト指向」の考え方に沿って、ストリームをモデル化したクラス群を標準ライブラリに用意して、入出力機能を実現している。(C 言語の場合は 入出力用の種々の関数が標準ライブラリ内に用意されていた。)

- `ostream` オブジェクトは、(言語処理系によって識別されたオブジェクトの型情報に従って) オブジェクトを適切な文字 (バイト) ストリームに変換する。



また、`istream` オブジェクトは、文字 (バイト) ストリームを適切な型のオブジェクトに変換する。



$\Longrightarrow$ **型安全**である。すなわち、入出力対象のデータ型を理解した上で常に適切な変換処理が為される。(一方、C 言語の `printf()` や `scanf()` では、式の型に合わない書式を指定して不可解な実行結果に至る可能性がある。)

- C 言語の printf() や scanf() との併用も可。**但し**、そのためには C 言語の cstdio ライブラリと C++言語の iostream ライブラリのそれぞれが保有しているバッファを同期させる必要があり、

```
ios::sync_with_stdio();
```

という関数呼び出しをライブラリを使う前に行なっておく。

- iostream ライブラリ内には、次の標準ストリームが用意されている。
  - cin ··· 標準入力
  - cout ··· 標準出力
  - cerr ··· 標準エラー出力 (バッファリングされない)
  - clog ··· 標準エラー出力 (バッファリングされる)
- iostream ライブラリ内では、通常は整数データを構成するビット列を左右にシフトするための演算子として使われる<< と >> を「**多重定義**」し、
  - ◇ << の左側に ostream オブジェクト (e.g.cout) が来た時には ostream オブジェクトが持っている出力機能を利用、
  - ◇ >> の左側に istream オブジェクト (e.g.cin) が来た時には istream オブジェクトが持っている入力機能を利用

  する様にしている。

書式を指定した出力： 演算子 << を用いて ostream オブジェクト (e.g.cout) に流し込まれたデータは、デフォルトでは表示に必要な最短の文字列となって出力される。出力の際の書式を設定したい場合は、設定したい書式に関連した**マニピュレータ (操作子)** を<< 演算子の右側に置く (i.e. データの代わりに ostream オブジェクトに流し込む) ことによって、ostream オブジェクトに指示を与える。(演算結果は、やはり第 1 オペランドとして指定された ostream オブジェクトである。マニピュレータを受けた ostream オブジェクトは、内部の動作設定を変える。) 具体的なマニピュレータ (操作子) としては、次の様なものが用意されている。(他にも多数ある。)

マニピュレータ	機能	マニピュレータが定義されているヘッダファイル
endl	バッファに溜まったデータと改行コードを順に出力先に吐き出す。	iostream
ends	ヌル文字を出力 (バッファに追加)。	iostream
flush	バッファに溜まったデータを出力先に吐き出す。	iostream
left	左寄せで出力する様に ostream オブジェクトの動作設定を変える。	iostream
right	右寄せで出力する様に ostream オブジェクトの動作設定を変える。	iostream
setw(n)	次の出力時に少なくとも n 桁で出力する様に ostream オブジェクトの動作設定を変える。(**注意：** 次の次以降の出力については、何の動作設定も行わない。)	iomanip
setfill(c)	空いた桁に文字 c を埋めて出力する様に ostream オブジェクトの動作設定を変える。	iomanip

(次ページに続く)

マニピュレータ	機能	マニピュレータが定義されているヘッダファイル
dec	整数の入出力を10進表記で行う様にostreamオブジェクトの動作設定を変える。	iostream
oct	整数の入出力を8進表記で行う様にostreamオブジェクトの動作設定を変える。	iostream
hex	整数の入出力を16進表記で行う様にostreamオブジェクトの動作設定を変える。	iostream
fixed	浮動小数点数を指数部なしの表記で出力する様にostreamオブジェクトの動作設定を変える。	iostream
scientific	浮動小数点数を小数点の前に1桁, 指数部付きの表記で出力する様にostreamオブジェクトの動作設定を変える。	iostream
setprecision(n)	浮動小数点数を小数点以下n桁の桁数で出力する様にostreamオブジェクトの動作設定を変える。	iomanip
boolalpha	bool型の値をtrueまたはfalseという形で出力する様にostreamオブジェクトの動作設定を変える。	iostream
noboolalpha	bool型の値を 1 または 0 という形で出力する様にostreamオブジェクトの動作設定を変える。	iostream

例 2.1 (sin(),cos(),tan()の表) それぞれの $k \in \{0, 1, 2, 3, ..., 12\}$ に対して $\sin(k\pi/12)$, $\cos(k\pi/12)$, $\tan(k\pi/12)$ の値を計算して、それらの結果を表の形に出力するC++プログラムを次に示す。

```
[motoki@x205a]$ cat -n printTableOfSinCosTan.cpp
     1 /* k=0,1,2,...,12として                          */
     2 /* sin(k*PI/12), cos(k*PI/12), tan(k*PI/12) */
     3 /* の表を出力するC++プログラム                 */
     4
     5 #include <iostream>
     6 #include <iomanip>
     7 #include <cmath>
     8 using namespace std;
     9
    10 const double PI = 3.1415926535897932;   // 円周率
    11
    12 int main()
    13 {
    14   cout << " k  sin(k*PI/12)  cos(k*PI/12)  tan(k*PI/12)" << endl
    15        << "--  ------------  ------------  ------------" << endl;
    16   for (int k=0; k<=12; k++) {
    17     cout << right
    18          << setw(2) << k
    19          << fixed << setprecision(9)
    20          << setw(14) << sin(k*PI/12.0)
```

```
    21          << setw(14) << cos(k*PI/12.0)
    22          << scientific << setprecision(5)
    23          << setw(14) << tan(k*PI/12.0) << endl;
    24   }
    25 }
[motoki@x205a]$ g++ printTableOfSinCosTan.cpp
[motoki@x205a]$ ./a.out
 k  sin(k*PI/12)  cos(k*PI/12)  tan(k*PI/12)
--  ------------  ------------  ------------
 0   0.000000000   1.000000000   0.00000e+00
 1   0.258819045   0.965925826   2.67949e-01
 2   0.500000000   0.866025404   5.77350e-01
 3   0.707106781   0.707106781   1.00000e+00
 4   0.866025404   0.500000000   1.73205e+00
 5   0.965925826   0.258819045   3.73205e+00
 6   1.000000000   0.000000000   1.63312e+16
 7   0.965925826  -0.258819045  -3.73205e+00
 8   0.866025404  -0.500000000  -1.73205e+00
 9   0.707106781  -0.707106781  -1.00000e+00
10   0.500000000  -0.866025404  -5.77350e-01
11   0.258819045  -0.965925826  -2.67949e-01
12   0.000000000  -1.000000000  -1.22465e-16
[motoki@x205a]$
```

ここで、

- プログラム6行目の#include指令は18~23行目でマニピュレータsetw(), setprecision()を使うので置いてある。
- プログラム7行目の#include指令は20~23行目でC言語の標準ライブラリ関数sin(), cos(), tan()を使うので置いてある。
- プログラム17行目のrightは、coutに対して「右揃えの出力」を指示するマニピュレータである。
- プログラム18行目,20~21行目,23行目に現れるsetw(2),setw(14)は、coutに対して「次の出力についての出力フィールドの最小幅(それぞれ2,14)」を指示するマニピュレータである。(指示された最小幅で足りない場合は、必要最小限のもっと広いフィールドに出力される。)
- プログラム19行目のfixedは、coutに対して「実数値についての指数部なしの出力」を指示するマニピュレータである。
- プログラム19行目,22行目のsetprecision(9),setprecision(5)は、coutに対して「実数値についての小数点以下の出力桁数(それぞれ9,5)」を指示するマニピュレータである。
- プログラム22行目のscientificは、coutに対して「実数値について小数点の前に1桁で指数部付きの出力」を指示するマニピュレータである。

istream オブジェクトのメンバ関数： istream オブジェクト (e.g.cin) は通常は>>演算子を通して使われるが、内部に次の様な関数も保有している。

関数プロトタイプ ··· 説明
`istream& get(char& c)` ··· 入力ストリームから次の1文字を取って来て変数 c に格納する。(**補足**：引数部に宣言されている"&"は、実引数として指定された変数に c という別名を付ける、すなわち参照呼出の形で引数結合するという指示を表す。)
`istream& get(char* s, int n, char c='\n')` ··· 入力ストリームから、引数 c で指定された区切り記号 (デフォルトでは改行コード) 又は入力終端 (EOF) までの文字の並び (但し長くなっても n-1 文字で打ち切り) を読み込み、最後に空文字 \0 を付けて char 型配列 s に格納する。
`istream& getline(char* s, int n, char c='\n')` ··· 上の引数3個の get(char*, int, char ='\n') とほぼ同じ。こちらの関数の場合は、最後の区切り記号は捨てられる。
`istream& read(char* s, int n)` ··· 指定した入力ストリームから、長さ n の文字の並び (但し途中で入力終端に至ればそこまで) を読み込み、char 型配列 s に格納する。

演算子>>を通した入力では空白類はデータ部を区切るための記号として読み飛ばされるため、場合によっては上記の関数も有用となる。また、入力の場合は、正しく入力されない可能性もあり、これらの事態に対処するために、次のメンバ関数も有用である。

関数プロトタイプ ··· 説明
`int good()` ··· 何の問題も起きてない (i.e.istream オブジェクト内の状態を表す変数に goodbit という値がセットされている) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
`int eof()` ··· 入力終端 (e.g. ファイルの終わり) に至っている (i.e.istream オブジェクト内の状態を表す変数に eofbit という値がセットされている) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
`int fail()` ··· 入力時に予期しない事態 (e.g. 数字が来るべき所で英字の入力) が発生した (i.e.istream オブジェクト内の状態を表す変数に failbit という値がセットされている) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
`int bad()` ··· 入力時に予期しない深刻な事態 (e.g. ディスク読み取りエラー) が発生した (i.e.istream オブジェクト内の状態を表す変数に badbit という値がセットされている) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
`int operator!()` ··· 入力時に予期しない事態 (もしくは深刻な事態) が発生した場合は非ゼロの値を返し、それ以外の場合は 0 を返す。正式の関数名は operator!() であるが、この関数は通常 !istream オブジェクト という風に演算子の形で使う。
(続く)

関数プロトタイプ　　･･･　　説明
int rdstate() 　･･･ istream オブジェクト内の状態を表す変数の値を返す。
void clear(int st=0) 　･･･ istream オブジェクト内の状態を表す変数の値を st に設定する。(st のデフォルト値である 0 は正常値 (goodbit) を表すので、実引数を省略して istream オブジェクト.clear() と書けば istream オブジェクトの状態をリセットすることになる。)
istream& ignore(streamsize n = 1, int delim = EOF) 　･･･ istream オブジェクト内のバッファに溜まっている文字を、delim というコード番号の文字が現れるまで、もしくは文字数が n に達するまで読み捨てる。

例 2.2 (円錐の体積) 入力ミスにも対処するようにして 2 つの実数データ r, h を読み込み、底面の半径が r、高さが h の円錐の体積を計算して出力する C++プログラムを次に示す。

```
[motoki@x205a]$ cat -n calculateVolumeOfCone.cpp
     1 /* 2つの実数データ r と h を読み込み、          */
     2 /*       底面の半径が r、高さが h の円錐の体積  */
     3 /* を出力する C++プログラム                     */
     4
     5 #include <iostream>
     6 #include <climits>
     7 using namespace std;
     8
     9 const double PI = 3.1415926535897932;   // 円周率
    10
    11 int main()
    12 {
    13   double radius, height;
    14
    15   cout << "底面の半径, 高さ : ";
    16   cin >> radius >> height;
    17   while (!cin) {
    18     cin.clear();
    19     cin.ignore(INT_MAX, '\n');
    20     cout << "[入力ミス --> 再度入力] 底面の半径, 高さ : ";
    21     cin >> radius >> height;
    22   }
    23
    24   cout << "底面の半径が " << radius
    25        << ", 高さが "     << height << " の円錐の体積" << endl
    26        << "     = " << PI*radius*radius*height/3.0 << endl;
    27 }
```

```
[motoki@x205a]$ g++ calculateVolumeOfCone.cpp
[motoki@x205a]$ ./a.out
底面の半径，高さ： 2.0 5.0
底面の半径が 2，高さが 5 の円錐の体積
     = 20.944
[motoki@x205a]$ ./a.out
底面の半径，高さ： 2.0 a
[入力ミス --> 再度入力] 底面の半径，高さ： a
[入力ミス --> 再度入力] 底面の半径，高さ： 2.0 5.0
底面の半径が 2，高さが 5 の円錐の体積
     = 20.944
[motoki@x205a]$
```

ここで、

- プログラム17~22行目が、入力ミスに対する対処のコードである。
- プログラム17行目の"!cin"は、cinに備わっている関数operator!()を演算子の形で呼び出したもので、「cinがエラー状態になっている」ということを表す。
- プログラム18行目の"cin.clear()"は、cinの状態を正常状態 (i.e. 状態を表す変数の値がgoodbit) に戻すことを表している。
- プログラム19行目の"INT_MAX"はC言語標準ライブラリclimits.h内で定義されたマクロで、int型で表せる最大整数を表す。従って、"cin.ignore(INT_MAX, '\n')"は「cin内のバッファに溜まっている文字を、改行コード('\n')が現れるまで読み捨てる」という意味になる。(バッファ内には (おそらく)INT_MAX個より多い文字は溜まってないだろう。)

2.9 演算子

{Pohl(1999)2.7節,C.9節, 柴田 (2009)p.76,p.143}

C言語からの変更：

- C言語の拡張。 (C言語で許されC++言語で許されないものはない。)
- C言語には無かったがC++言語で導入されたものは次の通り。
  - ◇ **スコープ解決演算子** :: ・・・これを使うと局所スコープで隠れた場所から大域変数にアクセスできる。⟶ 2.7節を参照。
  - ◇ **空き領域演算子** new ・・・C言語の標準ライブラリ関数malloc()の様に、空き領域から動的にメモリを確保する時に使う。⟶ 2.14節を参照。
  - ◇ **空き領域演算子** delete ・・・C言語の標準ライブラリ関数free()の様に、動的に確保したメモリを空き領域に返す時に使う。⟶ 2.14節を参照。
  - ◇ **キャスト演算子** データ型(式) ・・・古いC++で導入された関数記法のキャスト。⟶ 2.6節を参照。
  - ◇ **キャスト演算子** static_cast<データ型>(式) ・・・安全な型変換の場合に使う名前付きキャスト。⟶ 2.6節を参照。

◇ **キャスト演算子** `reinterpret_cast<`データ型`>(`式`)` ··· 安全でない型変換の場合に使う名前付きキャスト。⟶ 2.6 節を参照。

◇ **キャスト演算子** `const_cast<`データ型`>(`式`)` ··· const 性を変更したい場合に使う名前付きキャスト。⟶ 2.6 節を参照。

◇ **キャスト演算子** `dynamic_cast<`データ型`>(`式`)` ··· クラス階層上位のオブジェクトを下位オブジェクトと見做したい場合に使う名前付きキャスト。⟶ 2.6 節を参照。

◇ **メンバポインタ演算子** `.*` ··· クラス内の局所的なポインタ (オフセットの様なもの, **メンバポインタ**という) を通してオブジェクト内の要素にアクセスする時に使う。具体的に `x.*y` と書いた時、これは「オブジェクト x の中でメンバポインタ y を辿った先のメンバ」を表す。

◇ **メンバポインタ演算子** `->*` ··· クラス内の局所的なポインタ (i.e. メンバポインタ) を通してオブジェクト内の要素にアクセスする時に使う。具体的に `x->*y` と書いた時、これは「x の指すオブジェクトの中でメンバポインタ y を辿った先のメンバ」を表す。

◇ `typeid` **演算子** `typeid( )` ··· 引数の型が表すオブジェクトを作成する。`typeid(`引数`).name()` と書くと、これは引数の「型を表す文字列」(処理系に依存) の意味になる。

- C 言語の演算子を拡張 (多重定義) したものは次の通り。

◇ **挿入演算子** `<<` ··· ビット列の左シフトを表す演算子を拡張 (多重定義) し、演算子の左側に `ostream` オブジェクトが来た時はストリーム出力を表す様にした。⟶ 2.8 節を参照。

◇ **抽出演算子** `>>` ··· ビット列の右シフトを表す演算子を拡張 (多重定義) し、演算子の左側に `istream` オブジェクトが来た時はストリーム入力を表す様にした。⟶ 2.8 節を参照。

- C 言語と少し違っているものは次の通り。

◇ **関係演算子** `<`, `>`, `<=`, `>=` ··· `bool` 型が設けられたことに伴い演算結果が 0 または 1 ではなく `false` または `true` で表されるという違いのみ。本質的な違いはない。

◇ **論理演算子** `&&`, `||`, `!` ··· `bool` 型が設けられたことに伴う違いのみ。本質的な違いはない。

◇ **条件演算子** 条件 `?` 式 `:` 式 ··· `bool` 型が設けられたことに伴う違いのみ。本質的な違いはない。

◇ **後置の増分, 減分演算子** `++`, `--` ··· 優先順位が 1 つ上がった。

演算子の優先順位と結合性： 次の通り。

優先順位高 ↑

演算子	結合性
スコープ解決演算子 ::	左から右
関数の引数をくくる丸括弧 ( ) 配列添字をくくる四角括弧 [ ] メンバアクセス演算子 -> . ++ (後置) -- (後置) 関数表記のキャスト 型( ) static_cast< >( ) const_cast< >( ) dynamic_cast< >( ) reinterpret_cast< >( ) typeid( )	左から右
+ (単項) - (単項) ++ (前置) -- (前置) sizeof( ) ! ビット反転~ 間接演算子* 番地演算子& C 言語表記のキャスト 空き領域演算子 new delete	右から左
.* ->*	左から右
* / %	左から右
+ -	左から右
左シフトとストリームへの挿入 << 右シフトとストリームからの抽出 >>	左から右
< <= > >=	左から右
== !=	左から右
ビット積&	左から右
ビット排他的和^	左から右
ビット和|	左から右
&&	左から右
||	左から右
条件演算子 条件 ? 式 : 式	右から左
= += -= *= /= %= >>= <<= &= ^= |=	右から左
コンマ演算子,	左から右

2.10 制御構造

{Pohl(1999)2.8 節,C.10 節 }

C 言語の場合と同様の、次の制御構造が使える。

- if 文
  if-else 文
- while 文
- for 文
  (for ループの中の初期化式, 繰り返し式はコンマで区切った式のリストでもよい。)
- do-while 文
- break 文
- continue 文
- switch 文
- return 文
- 条件演算子 (式 1 ? 式 2 : 式 3)

- goto 文

2.11 関数

{Pohl(1999)p.31,p.69,3.5 節,3.7-8 節,6.2 節, 柴田 (1992)p.19}

(参考) メソッドに対する命名規則： Java 言語においては、メソッド (C/C++言語の「関数」に相当) に対する名前の付け方として、識別子名に対する一般指針 (2.2 節) に加えて、次の様な指針が一般的である。

- メソッド名の付け方：
  - ◇ 主として英小文字を用いる。
  - ◇ 2 つ以上の「意味のある単語」から構成する場合は、(単語の区切を明らかにするために)2 つ目以降の単語の頭文字を大文字にする。 Sun コーディング規約
  - ◇ メソッド名の場合 は、先頭文字を小文字にし、動詞 (句) を表す単語列にする。 Sun コーディング規約
  - ◇ フィールドの値を調べるメソッドの名前 は「"get"+フィールド名」にする。(e.g. getNum) 電通国際情報サービスコーディング規約
  - ◇ フィールドの値を設定するメソッドの名前 は「"set"+フィールド名」にする。(e.g. setValue) 電通国際情報サービスコーディング規約

main() の戻り値に関する C 言語との違い：

{ C++言語 ··· 暗黙の戻り値 0 を仮定
{ C 言語 ··· 暗黙の戻り値 なし

関数引数リストが空の場合の C 言語との違い：

{ C++言語 ··· 引数なしの意味
{ C 言語 ··· 引数の個数が不明で「引数チェックは行わない」という意味

関数の暗黙の実引数： 関数定義の際には、仮引数にデフォルト値を指定することができる。但し、

- デフォルト値を指定したい仮引数部については次の形で書く。

  データ型 仮引数名 = デフォルト値

- デフォルト値を指定する仮引数は、引数リストの後方に固める。(i.e. デフォルト値指定の仮引数の後ろに指定なしの仮引数を置くことは出来ない。)

  関数名(デフォルト値指定なし , デフォルト値指定あり)

  これにより、プログラム内で実引数列を指定して関数呼び出しを行った場合、
  ① 前から順に実引数と仮引数の対応が取られ、
  ② 対応する実引数がない仮引数についてはデフォルト値が設定される
  という風に、曖昧さなく引数間の対応付けを行うことができる。
- 同じスコープ内でデフォルト値指定を繰り返し書くことは出来ない。
- デフォルト値指定の記述は、ソースファイル上で、それを利用する関数呼び出しの前に配置する。

関数の inline 宣言： 関数定義の先頭に inline というキーワードを挿入することにより、コンパイラに

この関数を呼び出している場所に (関数呼び出しのコードではなく)
関数本体の処理コードを埋め込んで (i.e. **インラインに展開**して) もらいたい

という依頼を出すことができる。これに関して、

- inline 宣言されている関数であっても、関数の処理内容が複雑な場合 (e.g. 再帰関数) はコンパイラは必ずしもインラインに展開するとは限らない。
- inline 宣言によるインライン展開は**型安全**である。すなわち、プログラム中の関数呼び出しに対して、コンパイラは引数の (構文や) データ型を認識し、必要に応じて引数の適切な型変換を行った上でインライン展開を行う。
  一方、C言語で利用できる引数付きマクロを用いたマクロ展開は型安全ではなく、使い方を間違えると不可解な結果がもたらされることがある。例えば、

```
#define square(x) x*x
```

  という定義では、1.0/square(x) が 1.0/x*x と展開され、square(x+1) が x+1*x+1 と展開され、square(x++) が x++*x++ と展開されてしまう。また、

```
#define square(x) ((x)*(x));
```

  という風に定義を間違えると、コンパイル時にマクロを使用した場所での文法エラーとなってしまう。
  $\Longrightarrow$ **引数付きマクロの使用は避け、代わりにインライン関数を使用するのが望ましい。**

関数の多重定義： それぞれの関数には機能に合わせた適切な名前を付けるのが望まれるが、C言語では本質的に同じ機能でも関数ごとに別々の名前を付けることが求められている。こういった不都合を解消するために、C++言語では、次の条件を満たす場合に、定義する複数の関数の名前を同じに設定できる様になっている。

(条件) 仮引数の型のリスト (**シグネチャ**という) が互いに異なる。

同じ名前の関数が多重に定義されていても、コンパイラは、プログラム中の関数呼び出しに対して、実引数の型のリストと定義された関数のシグネチャ(i.e. 仮引数の型のリスト) を見比べ、最も適切な関数を割り出す。具体的には、

① 実引数の型のリストと合致する仮引数の型のリストが見つかれば、その仮引数の型のリストをもつ関数を選ぶ。
② 標準的な**格上げ** (promotion, e.g. float→double, bool→int, char→int,...) による型リストの合致を試みる。
③ 標準的な型変換による型リストの合致を試みる。
④ ユーザ定義の型変換による型リストの合致を試みる。
⑤ デフォルト実引数の可能性を探る。

例 2.3 (多重定義,inline 宣言) 引数で指定された値の最大値を返すmax() 関数群を多重定義したC++プログラムの例を次に示す。

```
[motoki@x205a]$ cat -n overloadMaxFunctions.cpp
     1 // max 関数群を多重定義した例
     2
     3 #include <iostream>
```

```
 4 using namespace std;
 5
 6 inline int max(const int x, const int y);
 7 inline int max(const int x, const int y, const int z);
 8 double max(const double a[], const int size);
 9
10 int main()
11 {
12   double a[5]={1.1, 9.9, 7.7, 3.3, 5.5};
13
14   cout << "max(1,8)=" << max(1,8) << endl;
15   cout << "max(1,33,8)=" << max(1,33,8) << endl;
16   cout << "max(a,5)=" << max(a,5) << endl;
17   cout << "::max(1.1,8.8)=" << ::max(1.1,8.8) << endl;
18 }
19
20 // 引数で指定された値の最大値を返す関数群
21 inline int max(const int x, const int y)
22 {
23   return (x<y) ? y : x;
24 }
25
26 inline int max(const int x, const int y, const int z)
27 {
28   return max(max(x, y), z);
29 }
30
31 double max(const double a[], const int size)
32 {
33   double max = a[0];
34   for (int i=1; i<size; i++) {
35     if (a[i] > max)
36       max = a[i];
37   }
38   return max;
39 }
[motoki@x205a]$ g++ overloadMaxFunctions.cpp
[motoki@x205a]$ ./a.out
max(1,8)=8
max(1,33,8)=33
max(a,5)=9.9
::max(1.1,8.8)=8
[motoki@x205a]$
```

ここで、

- プログラム17行目では、スコープ解決演算子"::"を用いてmaxという名前の有効範囲を明示し、呼び出し先の候補となる多重定義関数をソースプログラム内に限定している。このため、::max(1.1,8.8)という呼び出しに対して、実引数がint型に変換された上で21~24行目で定義されたmax(int,int)が呼び出されint型の値が返ってくる。これに対し、"::max(1.1,8.8)"ではなく単に"max(1.1,8.8)"としたのでは、標準ライブラリstd内で定義されたdouble max(double,double)が呼び出され、8ではなく8.8という戻り値が返ってくる。

2.12 名前空間

{Pohl(1999)3.10節, 柴田(2009)9.4節}

- 複数の所で作られたプログラムを合わせて使おうとした時、(関数名, 構造体名, クラス名,... 等の) 大域的な名前が衝突することがある。こういった不都合な事態を避けるために、局所的でない個々の宣言を特定の**名前空間**に所属させ、非局所的な名前を

  その名前の宣言された名前空間の名前 :: その名前

  という風に明示的に表す方式が導入された。
- 標準ライブラリの名前空間 std を暗黙の名前空間として扱いたい場合は、プログラムの最初の方に、

```
using namespace std;
```

  というusing指令を置けば良い。
- 個々のプログラマは、独自の名前空間を定義し、その中に種々の関数や変数定義, 構造体定義, クラス定義を入れることができる。例えば、

```
namespace English {
  int result;
  void printResult() {
    cout << "calculated value = " << result << endl;
  }
}
namespace Japanese {
  int result;
  void printResult() {
    cout << "計算結果 = " << result << endl;
  }
}
```

  という風に名前空間EnglishとJapaneseを定義しておけば、必要に応じてEnglish::result, English::printResult(), Japanese::result, Japanese::printResult()を区別して使うことができる。
- **無名の名前空間**の中で関数や外部変数を定義／宣言すれば、それらの関数や外部変数は、
  ◇ それが定義された同一ソースファイル内に限定される。
  ◇ 同一ソースファイル内では名前空間の指定無しで使用できる。
  例えば、

```
namespace {
  int result;
  void printResult() {
    cout << "calculated value = " << result << endl;
  }
}
```

と定義すれば、変数 result と関数 printResult() は名前空間の指定無しで同一ソースファイル内の残りの場所から (のみ) 参照できる。

$\Longrightarrow$ 外部変数や関数の有効範囲を同一ファイル内に制限したい場合は、宣言の前に static 修飾子を付ける他に、無名の名前空間を用いる方法がある。

> **補足:** "static"という修飾子は①関数内の局所変数に付けたり、②クラス内の要素である変数に付けたり、③外部変数や関数に付けたり、と様々な使い方がある。その中で、「外部変数や関数の有効範囲を制限する」ための③の用法は、他の使い方と全く異なる効果をもたらすので、C++言語では推奨されていない。

2.13 参照宣言

{Pohl(1999)3.12 節, 柴田 (2009)6.2 節 }

- 変数宣言できる場所で、特定の変数領域に別名 (エイリアス, **参照名**という) を付け、プログラム内でその「特定の変数領域」を表す名前や式の代わりに使うことができる。

> **補足:** 参照名の宣言によって新たなメモリ領域の確保を想定する訳ではない。それゆえ、メモリ領域を意味する「変数」という単語を避け「参照名」という言い方をしている。

- ブロック内、および関数外の変数宣言できる場所では、参照名の宣言は次の形で行う。

  データ型 & 参照名 = 参照先のメモリ領域を表す名前や式 ;

- 仮引数を宣言できる場所では、参照先の指定はせず参照名の宣言は次の形で行う。

  データ型 & 参照名

  この場合、参照名の表すメモリ領域は関数呼び出し時に動的に定まる。

  $\Longrightarrow$ 仮引数を参照名として宣言することにより参照呼び出しが実現される。

> **補足:** 基本的な考えとしては、参照名は新たなメモリ領域確保を想定するものではない。しかし、仮引数を参照名として宣言する場合は、参照先の動的な決定を実現するために、裏では対応する実引数へのポインタが関数に渡されることになる。
>
> $\Longrightarrow$ 関数内で参照仮引数の値の変更を行わないなら、その参照仮引数の宣言については const 修飾子も付けた方が良い。

例 2.4 (参照宣言) 参照宣言を使用した C++プログラムの例を次に示す。

```
[motoki@x205a]$ cat -n useReferenceDeclaration.cpp
     1 // 参照名を使用した例
     2
     3 #include <iostream>
```

```
 4 using namespace std;
 5
 6 int main()
 7 {
 8   int a[5] = {0, 10, 20, 30, 40};
 9   int& top = a[0];
10   int* p = a;
11
12   cout << "top+2=" << top+2 << endl;
13   cout << "*(p+2)=" << *(p+2) << endl;
14   cout << "*(&top+2)=" << *(&top+2) << endl;
15 }
[motoki@x205a]$ g++ useReferenceDeclaration.cpp
[motoki@x205a]$ ./a.out
top+2=2
*(p+2)=20
*(&top+2)=20
[motoki@x205a]$
```

ここで、

- プログラム 9 行目 では a[0] の参照名 (別名) として top という名前が宣言されている。従って、プログラム 12 行目,14 行目 中の、文字列リテラル外に現れる式 top+2, *(&top+2) はそれぞれ a[0]+2, *(&a[0]+2) という式と同等である。

例 2.5 (参照宣言を用いた swap() 関数の実装) 参照宣言を用いて引数で指定された変数の内容を交換する関数 swap() を実装したC++プログラムの例を次に示す。

```
[motoki@x205a]$ cat -n implementSwapByReference.cpp
 1 // 参照名を用いた swap(,) の実装
 2
 3 #include <iostream>
 4 using namespace std;
 5
 6 void swap(int& x, int& y);
 7
 8 int main()
 9 {
10   int a=0, b=5;
11
12   cout << "a=" << a << ", b=" << b << endl;
13   swap(a, b);
14   cout << "a=" << a << ", b=" << b << endl;
15 }
16
17 void swap(int& x, int& y)
```

```
    18 {
    19   int temp = x;
    20   x = y;
    21   y = temp;
    22 }
[motoki@x205a]$ g++ implementSwapByReference.cpp
[motoki@x205a]$ ./a.out
a=0, b=5
a=5, b=0
[motoki@x205a]$
```

2.14 空き領域演算子 new と delete

{Pohl(1999)3.20 節 }

空き領域演算子 new： **空き領域** (i.e. **ヒープ領域**) からメモリを確保するために、C 言語の標準ライブラリ関数 malloc(), calloc() の代わりになるものとして、単項演算子 new が用意されている。これに関して、

- この演算子は通常次のいずれかの形で使う。
  - ◇ new [データ型]
  - ◇ new [データ型] ([確保した領域に設定する初期値])
  - ◇ new [データ型] [[配列の要素数を表す式]]
- 上記 1~2 番目の書き方 をした場合は、指定されたデータ型の領域 1 個に相当するメモリがヒープ領域から確保され、この確保領域へのポインタが演算結果の値として返される。
- 上記 2 番目の書き方 をした場合は確保領域への初期設定も行われる。
- 上記 3 番目の書き方 をした場合は、指定されたデータ型と配列サイズをもつ配列領域がヒープ領域から確保され、この配列の先頭要素へのポインタが演算結果の値として返される。
- メモリ確保に失敗すると、bad_alloc 型の「例外オブジェクト」が発生したり、0 が演算結果として返されたりする。(**補足**：例外オブジェクトが発生した場合、発生した例外を処理するコードに制御が移り、そこでエラー処理が為される。)

空き領域演算子 delete： 空き領域 (i.e. ヒープ領域) から確保したメモリを開放するために、C 言語の 標準ライブラリ関数 free() の代わりになるものとして、単項演算子 delete が用意されている。これに関して、

- この演算子は次のいずれかの形で使う。
  - ◇ delete [確保領域へのポインタを保持する式]
  - ◇ delete [ ] [確保領域へのポインタを保持する式]
- 上記 1 番目の書き方 は配列領域以外を開放する時、2 番目の書き方 は配列領域を開放する時に使う。
- 演算結果の値はない。

例 2.6 (空き領域演算子 new,delete) 空き領域演算子 new と delete を使用した C++プログラムの例を次に示す。

```
[motoki@x205a]$ cat -n useFreeStoreOperations.cpp
     1 // 空き領域演算子 new と delete を使用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <cassert>
     6 using namespace std;
     7
     8 int main()
     9 {
    10   int* p = new int(333);
    11   int* data;
    12   int size;
    13
    14   cout << "*p = " << *p << endl;
    15
    16   //----------------------------
    17   cout << "配列サイズ： ";
    18   cin >> size;
    19   assert(size > 0);
    20
    21   data = new int[size];
    22   assert(data != 0);
    23
    24   for (int i=0; i<size; i++) {
    25     data[i] = i;
    26   }
    27
    28   cout << "配列 data の内容：";
    29   for (int i=0; i<size; i++) {
    30     if (i%5 == 0)
    31       cout << endl;
    32     cout << "  " << right << setw(5) << data[i];
    33   }
    34   cout << endl;
    35
    36   delete[] data;
    37 }
[motoki@x205a]$ g++ useFreeStoreOperations.cpp
[motoki@x205a]$ ./a.out
*p = 333
```

```
配列サイズ： 12
配列 data の内容：
      0      1      2      3      4
      5      6      7      8      9
     10     11
[motoki@x205a]$
```

ここで、

- プログラム 10 行目 は new 演算子の使い方を示すためだけのもので、変数の使い方としては不自然である。実際には、new 演算子を使ってヒープ領域から動的にメモリ確保するより、ブロック内で例えば
  ```
  int pp = 333;
  ```
  と変数宣言して*p の代わりに pp を使う方がずっと自然である。
- プログラム 19 行目,22 行目 に現れている assert() は C 言語の標準ライブラリ関数で、引数で与えられた条件を満たさない場合に実行を強制終了させる。例えば 19 行目 は、次の様に書く代わりの、簡易のエラーチェックのために配置している。

```
while (!cin || size<=0) {
  cin.clear();
  cin.ignore(INT_MAX, '\n');
  cout << "[入力ミス --> 再度入力] 配列サイズ： ";
  cin >> size;
}
```

2.15 抽象データ型 string と vector<>

{Pohl(1999)3.21 節, Stroustrup(2015, 第 4 版), シルト (2010)14.7 節, 柴田 (2009)p.369}

C++言語には様々な種類のライブラリが追加されている。例えば、ベクトルや連結リスト、集合、マップ (i.e. 写像のグラフ)、スタック、待ち行列、行列 (matrix) を
◇ 抽象データ型のデータとして表して扱ったり、
◇ 標準的なアルゴリズムで効率良く処理する
ためのライブラリ、エラー処理をサポートするためのライブラリ、数値計算のためのライブラリ、並行処理のためのライブラリ、等が提供されている。

ここでは、これらのライブラリの中から基本的なものを 2 つ選んで簡単に紹介する。

文字列を表すための抽象データ型 string：

- C 言語では 文字列を表すために、文字列の終端を表す文字 '\0' を最後に付加した文字の並びを char 型配列 (**ヌル終端文字配列**という) に保持する。これに関しては、
  ◇ 標準的な演算子を用いて処理できない。例えば、char s[80]; と宣言されていた時、s="string"; という代入文も"abc"+"def" という (連接) 演算も許されない。
  ◇ 安全性が保証されない。例えば、C 言語の標準ライブラリ関数 strcpy(), strcat() は文字列の格納先である配列に十分な容量があるかどうかのチェックを行わない。

この様な不都合を改善するために、文字列を表すための抽象データ型stringが追加された。

- 使う場合は #include <string> とする。
- 実体は basic_string<char> というクラス。

例 2.7 (抽象データ型string) 抽象データ型stringを使用したC++プログラムの例を次に示す。

```
[motoki@x205a]$ cat -n use_stringType.cpp
     1 // 抽象データ型stringを利用した例
     2
     3 #include <iostream>
     4 #include <string>
     5 #include <cctype>
     6 using namespace std;
     7
     8 int main()
     9 {
    10   string s1 = "abc", s2;
    11
    12   cout << "Enter a string: ";
    13   cin >> s2;
    14   s1 += s2;
    15
    16   cout << "s1 = " << s1 << endl;
    17   for (int i=0; i<s1.length(); i++) {
    18     s1[i] = toupper(s1[i]);
    19   }
    20   cout << "s1 = " << s1 << endl;
    21 }
[motoki@x205a]$ g++ use_stringType.cpp
[motoki@x205a]$ ./a.out
Enter a string: _def123
s1 = abc_def123
s1 = ABC_DEF123
[motoki@x205a]$
```

ここで、

- プログラム10行目の "abc" 自体はヌル終端文字配列で文字列を表す方式に沿っている。ここでは、このC言語標準方式で表された文字列リテラルを基に同じ文字列を表すstring型のオブジェクトを構成し変数s1に初期設定することになる。
- プログラム14行目の += は文字列の連接を表す。
- プログラム17行目の s1.length() は文字列s1の長さ (i.e. 構成要素数) を表す。

- プログラム 18 行目 の s1[i] は文字列 s1 中の添字番号 i の要素を表す。(添字番号が然るべき範囲に入っているかのチェックはしない。)
- プログラム 18 行目 の toupper() は C 言語の標準ライブラリ関数で、英字を大文字に変換する。関数プロトタイプが cctype.h に入っているので、5 行目 で#include している。

オブジェクトの列を表すための抽象データ型 vector<> :

- オブジェクトの列を表すために通常、配列が用いられる。しかし、
  - ◇ 配列の要素数はコンパイル時、もしくは new 演算子実行時に確定し、それ以降の実行の途中で変更することは出来ないので、不便に感じることもある。
  - ◇ 配列の要素数は属性として配列自身が保持している訳ではなく、別途パラメータ等で与えられるので、要素数の指定間違いといった単純なミスも起こり易い。

  この様な不便を改善し「**動的配列**」(i.e. サイズを動的に変えられる配列) を実現した抽象データ型として vector<要素の型> を利用できる。
- 使う場合は #include <vector> とする。

例 2.8 (抽象データ型 vector<int>) 抽象データ型 vector<int>を使用した C++プログラムの例を次に示す。

```
[motoki@x205a]$ cat -n use_vectorType.cpp
     1 // 抽象データ型 vector<int>を利用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <vector>
     6 using namespace std;
     7
     8 void showContentsOf(vector<int> v);
     9
    10 int main()
    11 {
    12   vector<int> v;
    13
    14   showContentsOf(v);
    15
    16   for (int i=0; i<10; i++)
    17     v.push_back(i);
    18   showContentsOf(v);
    19
    20   for (int i=0; i<5; i++)
    21     v.pop_back();
    22   showContentsOf(v);
    23
```

```
24   for (int i=0; i<v.size(); i++)
25     v[i] *= v[i];
26   showContentsOf(v);
27 }
28
29 void showContentsOf(vector<int> v)
30 {
31   cout << "要素数 = " << v.size() << endl;
32   cout << "内容   = (";
33   if (v.size() > 0)
34     cout << right << setw(2) << v[0];
35   for (int i=1; i<v.size(); i++)
36     cout << ", "<< right << setw(2) << v[i];
37   cout << " )" << endl;
38 }
[motoki@x205a]$ g++ use_vectorType.cpp
[motoki@x205a]$ ./a.out
要素数 = 0
内容   = ( )
要素数 = 10
内容   = ( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9 )
要素数 = 5
内容   = ( 0,  1,  2,  3,  4 )
要素数 = 5
内容   = ( 0,  1,  4,  9, 16 )
[motoki@x205a]$
```

ここで、

- プログラム12行目では、int型の値を要素とし要素数が0のvector<int>型オブジェクト(動的配列)が作られ、変数vの初期データとして設定される。
- プログラム29~38行目で定義され14行目,18行目,22行目,26行目で呼び出されている関数 showContentsOf() は引数で与えられた動的配列の大きさと内容を表示するためのものである。
- プログラム17行目の v.push_back(i) は、動的配列 v の末尾に値 i の要素を追加する操作を表す。
- プログラム21行目の v.pop_back() は、動的配列 v から末尾要素を削除する操作を表す。
- プログラム24行目,31行目,33行目,35行目の v.size() は、動的配列 v が保有する要素数を表す。

演習問題

□演習 2.1 (1000 以下の完全数) 正整数 k が等式

$$k = (k \text{ の約数の内、} k \text{ 以外のものの総和})$$

を満たす時、k は**完全数**であると言う。例えば、6 の約数は 1,2,3,6 の 4 個であり $6 = 1+2+3$ であるから、6 は完全数である。 1000 以下の完全数を全て出力する C++プログラムを作れ。

□演習 2.2 (ストリーム入出力, 冪乗) 有効桁が 10 桁の実数データ a を読み込み、$a^1, a^2, a^3, ..., a^{10}$ の値を表の形に見易く出力する C++プログラムを作成せよ。但し、ここでは

- 数学関数 pow() は使わないものとする。
- C 言語の標準ライブラリ関数は使わずに、C++言語で導入されたストリーム入出力を用いて次の形で出力せよ。

```
␣k␣␣␣␣␣␣␣␣a^k
--␣␣----------
␣1␣␣ | a^1 の値 |
␣2␣␣ | a^2 の値 |
 ⋮        ⋮
```

- 入力ミスにも対処するようにせよ。

□演習 2.3 (Fibonacci 数列) 一般に、初期値 a_0, a_1 と漸化式

$$a_n = a_{n-1} + a_{n-2} \quad \text{for each } n \geq 2$$

によって決まる数列 $\{a_n\}$ を **Fibonacci 数列**という。 初期値が $a_0 = a_1 = 1$ の Fibonacci 数列の第 1∼20 項目の値を計算し、それらの結果を表の形に出力する C++プログラムを作成せよ。

□演習 2.4 ((3n+1) 数列) 初期値 $a_1 = 100$ と漸化式

$$a_{n+1} = \begin{cases} 1 & \text{if } a_n = 1 \\ a_n/2 & \text{if } a_n \text{が偶数} \\ 3a_n + 1 & \text{otherwise} \end{cases}$$

によって定まる数列 $\{a_n\}$ の最初の 20 項 $a_1, a_2, a_3, ..., a_{20}$ を計算し、それらを表の形に見易く出力する C++プログラムを作成せよ。

□演習 2.5 (式の計算) 次の式の値を計算して出力する C++プログラムを作成せよ。

$$\sum_{i=0}^{18} \frac{(-1)^i}{i!} = \left(\left(\left(\cdots\left(\left(1 \cdot \frac{1}{-18} + 1\right)\frac{1}{-17} + 1\right)\cdots\right)\frac{1}{-3} + 1\right)\frac{1}{-2} + 1\right)\frac{1}{-1} + 1$$

□演習 2.6 (シンプソンの公式) シンプソンの公式によれば、定積分値は

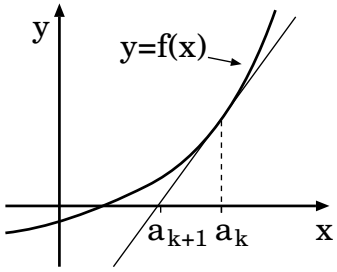
$$\begin{aligned} \int_a^b f(x)dx \approx \frac{h}{3}\Big[& f(a_0) + f(a_{2n}) \\ & +4(\, f(a_1) + f(a_3) + \cdots + f(a_{2n-3}) + f(a_{2n-1})\,) \\ & +2(\, f(a_2) + f(a_4) + \cdots + f(a_{2n-2})\,) \Big] \end{aligned}$$

但し、$h = \frac{b-a}{2n}$
$a_i = a + ih$
と近似でき、この近似の際の誤差は

$$|\text{誤差}| \leq \frac{(b-a)^5}{2880n^4} \max\left\{|f^{(4)}(\xi)| \;\middle|\; a{\leq}\xi{\leq}b\right\}$$

と見積もられる。$n = 1000$ としてこの公式を用いることによって $\int_0^1 \frac{4}{1+x^2}dx$ の近似値を計算する C++プログラムを作成せよ。

□演習 2.7 (Newton-Raphson 法)
一般に、方程式 $f(x) = 0$ の実根を数値的に求めるための方法として、Newton-Raphson 法は、適当な近似解 $x = a_1$ から出発して、漸近式 $a_{k+1} = a_k - f(a_k)/f'(a_k)$ により、次々とより良い近似解 $x = a_k (k = 1, 2, 3, ...)$ を求めていこうというものである。$f(a_k) = 0$ となった時点、あるいは $a_1, a_2, a_3, ...$ が十分に収束したと判断できる時点で、このアルゴリズムは終了させる。



方程式 $f(x) = x - \cos x = 0$ の場合は $f'(x) = 1 + \sin x$ であるから、

$$a_{k+1} = a_k - \frac{f(a_k)}{f'(a_k)} = a_k - \frac{a_k - \cos a_k}{1 + \sin a_k} \qquad (k = 1, 2, 3, ...)$$

によって近似解の改良を繰り返すことになる。初期近似解を $a_1 = 1.0$、終了条件を $|a_{k+1} - a_k| {\leq} 10^{-15}$ として Newton-Raphson 法を適用することによって、方程式 $f(x) = x - \cos x = 0$ の近似解 $x = 0.739\cdots$ を小数点以下 15 桁まで求める C++プログラムを作成せよ。

□演習 2.8 (平均と分散) n 個のデータ $x_0, x_1, x_2, \ldots, x_{n-1}$ の平均 μ と分散 V は数学的には

$$\begin{aligned} \mu &= \frac{1}{n}\sum_{i=0}^{n-1} x_i \\ V &= \frac{1}{n}\sum_{i=0}^{n-1} x_i^2 - \mu^2 \end{aligned}$$

と計算できる。これらの式に従って平均と分散を計算する C++プログラムを作成せよ。

□演習 2.9 (多重定義, 暗黙の実引数) 次の関数 `bubblesort()` を定義し、その動作テストを行う C++プログラムを作成せよ。

- 引数として `int` 型配列 `a` と `int` 型の値 `size` が与えられた時は、bubblesort アルゴリズムで配列要素 `a[0], a[1], ..., a[size-1]` を小さい順に並べ替える。
- 引数として `double` 型配列 `a` と `int` 型の値 `size` が与えられた時は、bubblesort アルゴリズムで配列要素 `a[0], a[1], ..., a[size-1]` を小さい順に並べ替える。
- 引数として `int` 型配列 `a` だけが与えられた時は、bubblesort アルゴリズムで配列要素 `a[0], a[1], ..., a[9]` を小さい順に並べ替える。
- 引数として `double` 型配列 `a` だけが与えられた時は、bubblesort アルゴリズムで配列要素 `a[0], a[1], ..., a[9]` を小さい順に並べ替える。

□**演習 2.10 (参照宣言)** 次の C++プログラムを実行するとどういう出力が得られるか？下の会話の様子中で [ ] の部分に予想される出力文字列を入れよ。但し、解答の際は空白を ␣ と明示せよ。下の会話の様子では、下線部はキーボードからの入力を表している。

```
[motoki@x205a]$ cat test1808_2a.cpp
#include <iostream>

using namespace std;

void f(int a);
void f(int* a);
void g(int& a);

int main()
{
  int a;

  a=0; f(a);
  cout << "(1)a=" << a << endl;
  a=0; f(&a);
  cout << "(2)a=" << a << endl;
  a=0; g(a);
  cout << "(3)a=" << a << endl;
}

void f(int a)  { a  += 100; }
void f(int* a) { *a += 100; }
void g(int& a) { a  += 100; }
[motoki@x205a]$ g++ test1808_2a.cpp
[motoki@x205a]$ ./a.out
[                                    ]
[motoki@x205a]$
```

オブジェクトベースプログラミング

3 C 言語構造体の考えの拡張、クラス

- C 言語の下での push-down スタックの実現
- 構造体内に内部データの操作方法も入れる, 構造体メンバに対するアクセス制御, クラス
- コンストラクタ, デストラクタ,static メンバ
- ソースファイルの構成
- 文法的な諸注意 (まとめ)

3.1 C 言語の下での push-down スタックの実現

{「プログラミング AI」(2018)13.2 節,Pohl(1992)3.3 節 }

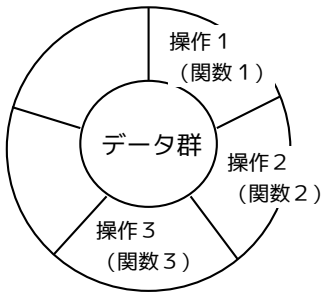
オブジェクト指向においては

> 関連するデータ群と操作 (関数) を 1 つにまとめて
> カプセル化し、ソフトウェア部品 (オブジェクト)
> として扱う

ということが基本である。これを実現するために C++言語に導入された機構を説明するに当たり、ここでは、
まず、試しに C 言語の下で

char 型データが最大 100 個入るスタック

をソフトウェア部品として提供することを考える。



実装例 3.1 (char 型データが最大 100 個入るスタック, 部品提供の実装案 1) 一般に、C 言語では、外部変数や関数に static 修飾子を付けると、それらの名前の有効範囲が同一ソースファイル内に限定される。それゆえ、
①1 つのソースファイル (カプセル) の中に関連するデータ群と操作 (関数) を入れ、
②隠蔽したいデータと関数に static 修飾子を付ければ、
このソースファイルはカプセル化され情報隠蔽されたソフトウェア部品として機能する。
以上の考えに基づいて「char 型データが最大 100 個入るスタック」部品を C ソースファイル (stackA_char100.c) の形で表した例を次に示す。

```
[motoki@x205a]$ cat -n stackA_char100.c
     1 /*************************************************************/
     2 /* char 型データが最大 100 個入るスタックを実装したモジュール  */
     3 /*-------------------------------------------                 */
     4 /*  外部へのサービスを行うために、次の 4 つの関数がこの       */
     5 /*  モジュールの中に用意されている。                          */
     6 /*  (1) スタックを空に初期化する関数 initializeStackA,       */
     7 /*  (2) スタックが空かどうかを調べる関数 isStackAEmpty,      */
     8 /*  (3) スタックに要素を 1 つ push-down する関数 pushdownIntoStackA, */
     9 /*  (4) スタックから要素を 1 つ pop-up する関数 popupFromStackA */
    10 /*************************************************************/
    11
```

```
12 #include <stdio.h>
13 #include <stdlib.h>
14 typedef  int  Boolean;
15
16 static char element[100];   /* モジュール外からは見えない */
17 static int  top;            /* モジュール外からは見えない */
18
19 /* 外部へのサービスを行うための関数群 */
20 void initializeStackA(void)
21 {
22   top = -1;
23 }
24
25 Boolean isStackAEmpty(void)
26 {
27   return (top < 0);
28 }
29
30 void pushdownIntoStackA(char c)
31 {
32   if (++top >= 100) {
33     printf("stack overflow\n");
34     exit(1);
35   }
36   element[top] = c;
37 }
38
39 char popupFromStackA(void)
40 {
41   if (top < 0) {
42     printf("popup from empty stack\n");
43     exit(1);
44   }
45   return element[top--];
46 }
[motoki@x205a]$
```

これに関して、

- このソフトウェア部品を利用した例を次に示す。

```
[motoki@x205a]$ cat -n reverseWordThroughStack1.c
     1 /* 文字列を 1 個読み込み、                                    */
     2 /* それをスタックを用いて反転した後出力する C プログラム */
     3 /* (スタックを 1 個のソースファイルで実装する版)          */
     4
```

```
 5 #include <stdio.h>
 6 typedef  int  Boolean;
 7
 8 void initializeStackA(void);
 9 Boolean isStackAEmpty(void);
10 void pushdownIntoStackA(char c);
11 char popupFromStackA(void);
12
13 int main(void)
14 {
15   char s[100];
16   int  i;
17
18   printf("Input a string:  ");
19   scanf("%s", s);
20   initializeStackA();
21   for (i=0; s[i]!='\0'; ++i)
22     pushdownIntoStackA(s[i]);
23   for (i=0; !isStackAEmpty(); ++i)
24     s[i] = popupFromStackA();
25   printf("Reversed string: %s\n", s);
26   return 0;
27 }
[motoki@x205a]$ gcc reverseWordThroughStack1.c stackA_char100.c
[motoki@x205a]$ ./a.out
Input a string:  abc123
Reversed string: 321cba
[motoki@x205a]$
```

- このソフトウェア部品提供方式の利点・欠点は次の通り。
  (**利点**) スタック領域への操作は用意された 4 つの関数を通してのみ可能なので、スタックの特性である**「後入れ先出し」(LIFO) の原則が保証**される。
  (**欠点**) ソフトウェア部品 (オブジェクト) を直接ソースファイルの形で提供するので、**同種の部品が複数必要な場合にうまく対処できない。**(必要な分だけ類似のソースファイルを用意する必要がある。)

実装例 3.2 (char 型データが最大 100 個入るスタック, 部品提供の実装案 2) 先の実装例 3.1 の欠点を解消し、必要なだけ簡単にスタックをソフトウェア部品として生成できる様にするための C 言語の 方法としては、構造体の利用が考えられる。この方向での実装案として、
①スタック領域と top の番号を保持する領域をメンバにもつ構造体の枠組みの定義と、
②関連する操作 (関数) 群
を 1 つのファイルにまとめた例を次に示す。

```
[motoki@x205a]$ cat -n struct_stackChar100.h
```

```
 1 /* 「char 型データが最大 100 個入るスタック」を表す構造体の枠組みと */
 2 /* これらの構造体を操作するための関数群                          */
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 typedef int Boolean;
 7
 8 struct stackChar100 {
 9   char element[100];
10   int  top;
11 };
12
13 /* StackChar100 型構造体を操作するための関数群 */
14 void initialize(struct stackChar100 *stack)
15 {
16   stack->top = -1;
17 }
18
19 Boolean isEmpty(struct stackChar100 *stack)
20 {
21   return (stack->top < 0);
22 }
23
24 void pushdown(struct stackChar100 *stack, char c)
25 {
26   if (++stack->top >= 100) {
27     printf("stack overflow\n");
28     exit(1);
29   }
30   stack->element[stack->top] = c;
31 }
32
33 char popup(struct stackChar100 *stack)
34 {
35   if (stack->top < 0) {
36     printf("popup from empty stack\n");
37     exit(1);
38   }
39   return stack->element[stack->top--];
40 }
[motoki@x205a]$
```

これに関して、

- このソフトウェア部品提供の枠組みを利用した例を次に示す。

```
[motoki@x205a]$ cat -n reverseWordThroughStack2.c
     1 /* 文字列を 1 個読み込み、                                    */
     2 /* それをスタックを用いて反転した後出力する C プログラム */
     3 /* (スタックを構造体で実装する版)                          */
     4
     5 #include <stdio.h>
     6 #include <stdlib.h>
     7 #include "struct_stackChar100.h"
     8
     9 int main(void)
    10 {
    11   struct stackChar100 stack;
    12   char s[100];
    13   int  i;
    14
    15   printf("Input a string:  ");
    16   scanf("%99s", s);
    17   initialize(&stack);
    18   for (i=0; s[i]!='\0'; ++i)
    19     pushdown(&stack, s[i]);
    20   for (i=0; !isEmpty(&stack); ++i)
    21     s[i] = popup(&stack);
    22   printf("Reversed string: %s\n", s);
    23   return 0;
    24 }
[motoki@x205a]$ gcc reverseWordThroughStack2.c
[motoki@x205a]$ ./a.out
Input a string:  abc123
Reversed string: 321cba
[motoki@x205a]$
```

- このソフトウェア部品提供方式の利点・欠点は次の通り。
  (利点) 同種の部品が複数必要な場合にもうまく対処できる。(単に、構造体を表す変数を宣言するだけ。)
  (欠点) スタック (構造体) を操作するために用意された 4 つの関数だけを使う場合は問題ないが、スタック (構造体) 内のデータは全て直接の操作が可能である。従って、スタックの特性である**「後入れ先出し」(LIFO) の原則は保証されない。**

3.2 C 言語構造体の考えの拡張、クラス

{Pohl(1992)3.4-7 節 }

先の実装例 3.1∼ 3.2 から、もし
① 構造体のカプセルの中に構造体内部のデータを操作する関数を入れることができ、また
② 必要に応じて構造体メンバの隠蔽もできる

様になっていれば、生成されたそれぞれの構造体は実装例 3.1 で考えたソースファイル stackA_char100.c と同様に、カプセル化され情報隠蔽されたソフトウェア部品として機能する、と考えられる。この様な考えの下、実際に C++言語では構造体に関する上記①~②の拡張が行われている。

実装例 3.3 (char 型データが最大 100 個入るスタック, 部品提供の実装案 3) C++言語の下で、上記①~②の拡張に沿って実装例 3.2 の構造体定義を書き直した例を次に示す。

```
[motoki@x205a]$ cat -n StackChar100_ver0struct.h
     1 /* 「char 型データが最大 100 個入るスタック」を表す構造体の枠組み */
     2
     3 #include <iostream>
     4 #include <cstdlib>
     5
     6 struct StackChar100 {
     7 private:
     8   char element[100];
     9   int  top;
    10 public: //StackChar100 型構造体を操作するための関数群
    11   void initialize() { top = -1; }
    12   bool isEmpty() { return (top < 0); }
    13   void pushdown(char c);
    14   char popup();
    15 };
    16
    17 void StackChar100::pushdown(char c)
    18 {
    19   if (++top >= 100) {
    20     std::cout << "stack overflow" << std::endl;
    21     exit(1);
    22   }
    23   element[top] = c;
    24 }
    25
    26 char StackChar100::popup()
    27 {
    28   if (top < 0) {
    29     std::cout << "popup from empty stack" << std::endl;
    30     exit(1);
    31   }
    32   return element[top--];
    33 }
[motoki@x205a]$
```

これに関して、

- C++言語では構造体カプセルの中に関数を入れることができるので、プログラム 11~14 行目 では構造体の構成要素としたい関数を並べている。
- 構造体メンバとした関数の内、処理内容が非常に単純なものについては、11~12 行目 の様に関数定義を丸ごと構造体定義の中に配置している。これらの関数は (暗黙に)inline 宣言されたものとして扱われ、(もし可能であれば、) コンパイル時にこれらの関数の呼び出し場所に関数呼び出しのコードではなく関数本体の処理コードが埋め込まれる。
- 構造体メンバとした関数の内、処理内容が 1 つの文で表せないものについては、struct ...{...} の部分が長くなって構造体の構成要素の見通しが悪くなるのを避けるため、13~14 行目 の様に struct 構文の中には関数プロトタイプだけを配置し、本体も含めた関数定義は struct 構文の外 (17~33 行目) に配置している。
- プログラム 17 行目,26 行目 の StackChar100::pushdown, StackChar100::popup では、pushdown, popup という名前が構造体 StackChar100 のメンバ関数の名前であることを**スコープ解決演算子**:: を用いて明示している。
- struct 構文の外 (17~33 行目) で定義された 2 つの関数については、inline 宣言されていないので、コンパイル時にインライン展開されることはなく、関数の呼び出し場所に関数呼び出しのコードが埋め込まれる。
- プログラム 7 行目 の private:は、これ以降 (別の指示があるまで) のメンバを**非公開**にし構造体外からのアクセスを禁止することを宣言している。
- プログラム 10 行目 の public:は、これ以降 (別の指示があるまで) のメンバを**公開**し構造体外からのアクセスを許可することを宣言している。
- プログラム 20 行目,29 行目 の std:は、cout と endl が標準ライブラリ内で定義された名前であることを明示するために付けている。

> **補足:** ヘッダファイル内に「using namespace std;」という行を入れると、このヘッダファイルをインクルードするソースファイル上でも「using namespace std;」という using 指令が (知らない内に) 有効になってしまう。そこで、ヘッダファイルの汎用性を保つため、ここでは using 指令の使用を避けている。

- このソフトウェア部品提供の枠組みを利用した例を次に示す。

```
[motoki@x205a]$ cat -n reverseWordThroughStack3.cpp
     1 /* 文字列を 1 個読み込み、                                    */
     2 /* それをスタックを用いて反転した後出力する C++プログラム */
     3 /* (スタックを C++構造体で実装する版)                       */
     4
     5 #include <iostream>
     6 #include <string>
     7 #include "StackChar100_ver0struct.h"
     8 using namespace std;
     9
    10 int main()
    11 {
    12   StackChar100 stack;
```

```
13   string s;
14
15   cout << "Input a string:  ";
16   cin >> s;
17   stack.initialize();
18   for (int i=0; i < s.length(); ++i)
19     stack.pushdown(s[i]);
20   for (int i=0; !stack.isEmpty(); ++i)
21     s[i] = stack.popup();
22   cout << "Reversed string: " << s << endl;
23 }
[motoki@x205a]$ g++ reverseWordThroughStack3.cpp
[motoki@x205a]$ ./a.out
Input a string:  abc123
Reversed string: 321cba
[motoki@x205a]$
```

ここで、

◇ 利用例のプログラム 12 行目 に見られる様に、C++言語では **構造体タグ**をデータ型名として用いることができる。

- このソフトウェア部品提供方式の利点・欠点は次の通り。

  (利点) 同種の部品が複数必要な場合にもうまく対処できる。(単に、構造体を表す変数を宣言するだけ。)

  (利点) スタック領域への操作は用意された 4 つの関数を通してのみ可能なので、スタックの特性である**「後入れ先出し」(LIFO) の原則が保証**される。

実装例 3.4 (char 型データが最大 100 個入るスタック, 部品提供の実装案 4) スタックをカプセル化され情報隠蔽されたソフトウェア部品として提供するに当たって、先の実装例 3.3 では、目的とするソフトウェア部品 (オブジェクト) の設計図を struct 構文を用いて定義できることを示した。同等の定義は次に示す様に class 構文を用いて行うこともできる。

```
[motoki@x205a]$ cat -n StackChar100_ver1.h
 1 /* 「char 型データが最大 100 個入るスタック」を表すオブジェクトの枠組み */
 2
 3 #include <iostream>
 4 #include <cstdlib>
 5
 6 class StackChar100 {
 7   char element[100];
 8   int  top;
 9 public: //StackChar100 型構造体を操作するための関数群
10   void initialize() { top = -1; }
11   bool isEmpty() { return (top < 0); }
12   void pushdown(char c);
```

```
13   char popup();
14 };
15
16 void StackChar100::pushdown(char c)
17 {
18   if (++top >= 100) {
19     std::cout << "stack overflow" << std::endl;
20     exit(1);
21   }
22   element[top] = c;
23 }
24
25 char StackChar100::popup()
26 {
27   if (top < 0) {
28     std::cout << "popup from empty stack" << std::endl;
29     exit(1);
30   }
31   return element[top--];
32 }
[motoki@x205a]$
```

これに関して、

- 実装例 3.3 で示した StackChar100_ver0struct.h との違いは、(コメント以外では) 次の 2 点だけである。
  - ◇ プログラム 6 行目 のキーワードが struct から class に変更された。
  - ◇ プログラム 6 行目 の次の行にあった private:宣言が無くなった。
- キーワード struct を使った場合はメンバは原則公開でオブジェクト外からのアクセスは暗黙に許可される。しかし、キーワード class を使った場合はメンバは原則非公開でオブジェクト外からのアクセスが暗黙に禁止されるので、上のプログラムでは 6 行目 の次に private:宣言を置くのを省略している。

> **補足:** 2 つのキーワード struct と class の効果の違いは、オブジェクト外からのアクセスに関する暗黙の設定 (公開／非公開) がどうなるかということだけである。
> $\Longrightarrow$ プログラム 6~32 行目 では、結局は構造体の枠組み／設計図が定義されていることになる。

一般に、C++言語の struct 構文や class 構文もしくは、これに相当する形で定義されるソフトウェア部品 (オブジェクト) の枠組み／設計図／種類のことを**クラス**と呼ぶ。また、定義されたクラスに対して、そのクラスに属するソフトウェア部品のことをそのクラスの**インスタンス**と呼ぶ。

3.3 コンストラクタとデストラクタ, 静的メンバ

{Pohl(1992)4.1-2 節,Pohl(1999)5.2-3 節, 柴田 (2009)11 章,13 章 }

一般にソフトウェア部品は使う前に然るべき初期化を行うべきであるので、C++言語では、クラスの設計図の基づいたソフトウェア部品 (オブジェクト) を生成する際に初期化のために自動的に呼び出される関数 (**コンストラクタ**という) を用意できる様になっている。また、クラスの設計図の基づいて生成されたオブジェクトの中にはヒープ領域から確保された領域へのポインタを含むこともあるので、ブロック終了等に伴うオブジェクト消滅の際に後始末 (e.g. 領域開放) のために自動的に呼び出される関数 (**デストラクタ**という) も用意できる様になっている。コンストラクタの名前はクラス名と同じ、デストラクタの名前はクラス名の前にティルダ記号 (~) を配置した名前 (i.e.~クラス名)、と決まっている。

実装例 3.5 (char 型データが入るスタック, 部品提供の実装案 5) 実装例 3.4 のクラス定義にコンストラクタとデストラクタを加えてスタックの容量をオブジェクト生成時に設定できる様にした例を次に示す。

```
[motoki@x205a]$ cat -n StackChar_ver1.h
     1 /* 「char 型データが入るスタック」を表すオブジェクトの枠組み */
     2
     3 #include <iostream>
     4 #include <cstdlib>
     5
     6 class StackChar {
     7   char* element;
     8   int  size;
     9   int  top;
    10 public:
    11   explicit StackChar(int size = 100);
    12   ~StackChar() { delete[] element; }
    13   //StackChar 型オブジェクトを操作するための関数群
    14   bool isEmpty() const { return (top < 0); }
    15   void pushdown(char c);
    16   char popup();
    17 };
    18
    19 StackChar::StackChar(int size): size(size), top(-1)
    20 {
    21   element = new char[size];
    22 }
    23
    24 void StackChar::pushdown(char c)
    25 {
    26   if (++top >= size) {
```

```
27      std::cout << "stack overflow" << std::endl;
28      exit(1);
29    }
30    element[top] = c;
31  }
32
33  char StackChar::popup()
34  {
35    if (top < 0) {
36      std::cout << "popup from empty stack" << std::endl;
37      exit(1);
38    }
39    return element[top--];
40  }
[motoki@x205a]$
```

これに関して、

- インスタンス生成時のスタック容量設定を可能にするために、データを入れる領域はヒープ領域から確保する様にした。プログラム 7 行目 の element はこの確保領域へのポインタ、8 行目 の size はこの確保領域の容量を保持する。
- プログラム 11 行目,19~22 行目 がコンストラクタを記述した部分、12 行目 がデストラクタを記述した部分である。コンストラクタとデストラクタについては戻り値は常に無いので、関数宣言時に戻り値の型として何も書かない。(void と書くこともしない。)
- 引数が 1 個のコンストラクタはデフォルトでは「引数の型 → コンストラクタが扱うオブジェクトの型」という型変換に使われる。プログラム 11 行目 の explicit 修飾子はこのコンストラクタを型変換に使用しないことを宣言している。
- プログラム 19 行目 の「: size(size), top(-1)」の部分は**初期化子リスト**と呼ばれるもので、メンバの初期化をどう行うかを指示している。丸括弧の前の名前 size, top がメンバ名を表し、丸括弧の中の式 size, -1 が初期設定値を表す。

> **補足:** メンバが基本データ型でなく、何らかのクラス定義に従ったオブジェクトである場合には
> メンバ名 (コンストラクタへの実引数列)
> という形で初期設定の指定を書ける。

- プログラム 19~22 行目 は次の様に書くのと同じである。

```
StackChar::StackChar(int size)
{
  this->size = size;
  element = new char[size];
  this->top = -1;
}
```

補足: クラス定義に従ったオブジェクトであるメンバへの初期設定を関数本体内で行いたい場合は、コンストラクタを明示的に呼び出して

メンバ名=クラス名(コンストラクタへの実引数列);

という形で初期設定の指定を書ける。しかし、この書き方ではクラス名型のオブジェクトが一時的に作られそれが変数メンバ名に代入されるので、処理が非効率になる。

- プログラム14行目のconst宣言は関数本体内でデータメンバの値を変更しないことを宣言している。
- このソフトウェア部品提供の枠組みを利用した例を次に示す。

```
[motoki@x205a]$ cat -n reverseWordThroughStack5.cpp
     1 /* 文字列を1個読み込み、                                        */
     2 /* それをスタックを用いて反転した後出力するC++プログラム */
     3 /* (スタックをコンストラクタ付きC++クラスのインスタンスとして実装) */
     4
     5 #include <iostream>
     6 #include <string>
     7 #include "StackChar_ver1.h"
     8 using namespace std;
     9
    10 int main()
    11 {
    12   StackChar stack(100);
    13   string s;
    14
    15   cout << "Input a string:  ";
    16   cin >> s;
    17   for (int i=0; i < s.length(); ++i)
    18     stack.pushdown(s[i]);
    19   for (int i=0; !stack.isEmpty(); ++i)
    20     s[i] = stack.popup();
    21   cout << "Reversed string: " << s << endl;
    22 }
[motoki@x205a]$ g++ reverseWordThroughStack5.cpp
[motoki@x205a]$ ./a.out
Input a string:  abc123
Reversed string: 321cba
[motoki@x205a]$
```

ここで、

◇ 利用例のプログラム12行目の stack(100) は実引数列が (100) のコンストラクタ StackChar(100) を呼び出し、stack というメンバ名の付いた StackChar オブジェクトの初期設定を行うことを指示している。

補足: この場合は
```
StackChar stack = StackChar(100);
```
という書き方も可能である。しかし、この書き方ではコンストラクタの指示に従った StackChar 型のオブジェクトが一時的に作られそれが変数 stack に代入されるので、処理が非効率になる。

実装例 3.6 (char 型データが入るスタック, 部品提供の実装案 6) クラス定義は、目先の使用だけでなく、将来の使用も念頭に置いて行うべきである。そこで、実装例 3.5 のクラス定義に幾つかの有用そうなメンバを加えてクラスの拡充を図った例を次に示す。

```
[motoki@x205a]$ cat -n StackChar_ver2.h
     1 /* 「char 型データが入るスタック」を表すオブジェクトの枠組み */
     2
     3 #include <iostream>
     4 #include <string>
     5 #include <cstdlib>    // for exit()
     6 #include <cstring>    // for memcpy()
     7
     8 class StackChar {
     9   static int numOfInstances;
    10   const int id;
    11   char* element;
    12   int  size;
    13   int  top;
    14 public:
    15   explicit StackChar(int size = 100);//デフォルトコンストラクタを含む
    16   StackChar(const StackChar& stack);  // コピーコンストラクタ
    17   StackChar(const std::string& str);
    18   StackChar(int size, const std::string& str);
    19   ~StackChar() { delete[] element; }
    20   // オブジェクト (や StackChar クラス全体) の情報を提供するための関数群
    21   static int getNumOfInstances() { return numOfInstances; }
    22   int getId() const { return id; }
    23   int getSize() const { return size; }
    24   int getNumOfElements() const { return top+1; }
    25   void showContents() const;
    26   // StackChar 型オブジェクトを操作するための関数群
    27   void reset() { top = -1; }
    28   void resize(int size);
    29   bool isEmpty() const { return (top < 0); }
    30   void pushdown(char c);
    31   char popup();
    32 };
    33
```

```
34 // static 変数の初期化 ----------------------------------------
35 int StackChar::numOfInstances = 0;
36
37 // 各種コンストラクタ ------------------------------------------
38 StackChar::StackChar(int size)     // デフォルトコンストラクタを含む
39           : id(numOfInstances++), size(size), top(-1)
40 {
41   element = new char[size];
42 }
43
44 StackChar::StackChar(const StackChar& stack) // コピーコンストラクタ
45           : id(numOfInstances++), size(stack.size), top(stack.top)
46 {
47   element = new char[stack.size];
48   memcpy(element, stack.element, stack.size);
49 }
50
51 StackChar::StackChar(const std::string& str)
52           : id(numOfInstances++), size(str.length()), top(str.length()-1)
53 {
54   element = new char[size];
55   for (int i=0; i<str.length(); i++)
56     element[i] = str[i];
57 }
58
59 StackChar::StackChar(int size, const std::string& str)
60           : id(numOfInstances++), size(size), top(str.length()-1)
61 {
62   element = new char[size];
63   for (int i=0; i<str.length(); i++)
64     element[i] = str[i];
65 }
66
67 // オブジェクトの情報を提供するための関数群 ----------------------------
68 void StackChar::showContents() const    // スタックの内容を表示
69 {
70   std::cout << "stack_of_char(id=" << id << ",size=" << size << ") has "
71             << top+1 << " elements as follows:" << std::endl;
72   std::cout << "  [Bottom] ";
73   for (int i=0; i<=top; i++)
74     std::cout << element[i] << "  ";
75   std::cout << "<--" << std::endl;
76 }
```

```
 77
 78 // StackChar型オブジェクトを操作するための関数群 ----------------------
 79 void StackChar::resize(int size)
 80 {
 81   if (top >= size) {
 82     std::cout << "insufficient stack size" << std::endl;
 83     exit(1);
 84   }
 85   char* temp = new char[size];
 86   memcpy(temp, element, top+1);
 87   delete[] element;
 88   this->size = size;
 89   element = temp;
 90 }
 91
 92 void StackChar::pushdown(char c)                    // pushdown 操作
 93 {
 94   if (++top >= size) {
 95     this->resize(size+10);
 96   }
 97   element[top] = c;
 98 }
 99
100 char StackChar::popup()                              // popup 操作
101 {
102   if (top < 0) {
103     std::cout << "popup from empty stack" << std::endl;
104     exit(1);
105   }
106   return element[top--];
107 }
[motoki@x205a]$
```

これに関して、

- 先の実装例 3.5 のクラス定義に追加／加筆されたのは主に次の要素である。
  - ◇ 9 行目,35 行目 numOfInstances ... これまでに生成した StackChar インスタンスの個数
  - ◇ 10 行目 id ... 個々の StackChar インスタンスに固有の id 番号
  - ◇ 16∼18 行目,44∼65 行目 ... コンストラクタ
  - ◇ 21∼24 行目 get...() ... id の値, size の値, スタック内のデータ数の情報を提供
  - ◇ 25 行目,68∼76 行目 showContents() ... スタック内部の状況を表示
  - ◇ 27 行目 reset() ... スタックを空にリセット
  - ◇ 28 行目,79∼90 行目 resize() ... スタックの容量を変更
  - ◇ 30 行目,92∼98 行目 pushdown() ... スタック満杯の場合は resize() を用いて容量

を増やして対応する様に変更

- プログラム 9 行目 の static は、numOfInstances が個々のインスタンスの保有するメンバではなくクラス全体で保有するメンバ (**静的メンバ**という) であることを宣言している。(これまでに生成した StackChar インスタンスの個数を個々のインスタンスが保有する訳にも行かない。) クラス外からこの静的メンバにアクセスしたい場合は、StackChar.numOfInstances ではなく StackChar::numOfInstances と書く。
- 当然、静的メンバ numOfInstances はクラス定義に伴って領域確保される変数で、その初期化を行なっているのが 35 行目 である。一般に、静的メンバ変数の初期化はクラス定義の中で行うことはできず、クラス定義の外で static 修飾子も付けずに

  データ型 クラス名 :: 静的メンバ変数名 = 初期値を表す式 ;

  という形で行う。
- プログラム 10 行目 の id はインスタンスに固有の id 番号を表し、上の numOfInstances の情報に基づいてインスタンス生成時に割り振られる。一旦決まった id 番号は変更されるべきでないので、10 行目 の先頭に const 修飾子を付けている。
- 一般に、引数なしで呼び出されるコンストラクタは**デフォルトコンストラクタ**と呼ばれ、当該クラスに属するオブジェクトを保持する変数や配列の宣言の際に初期設定の指定がない場合に暗黙に呼び出される、という特別な役割を果たす。上で定義したクラス StackChar の場合、StackChar() は 15 行目,38~42 行目 のコンストラクタに合致するので、このコンストラクタがデフォルトコンストラクタの役割を果たすことになる。
- 一般に、同クラスに属する別インスタンスをコピーすることによってオブジェクトの初期化を行うコンストラクタを**コピーコンストラクタ**と呼ぶ。上で定義したクラス StackChar の場合は、16 行目,44~49 行目 のコンストラクタがコピーコンストラクタである。
- 17~18 行目,51~65 行目 のコンストラクタ (2 種) は、引数で与えられた string 型文字列内の文字を順にスタックに積むことによって StackChar インスタンスの初期化を行う。
- プログラム 21 行目 の static は、この行で宣言しているメンバ関数 getNumOfInstances() が個々のインスタンスに備わっているものではなく、クラス全体に備わったもの (i.e. 静的メンバ) であることを宣言している。クラス外から参照したい場合は StackChar::getNumOfInstances() という書き方をする。
- プログラム 22~24 行目 のメンバ関数 getId(), getSize(), getNumOfElements() は、それぞれメンバ id の値, メンバ size の値, スタック内に溜まったデータ数を外部に答える働きをする。インスタンス内のデータの変更を行わないので処理の本体部の前で const 宣言している。(これらの関数の名前は Java 言語の命名規則に従っている。)
- プログラム 48 行目,86 行目 の memcpy() は、char 型配列内のバイト列を丸ごと別の char 型配列にコピーする C 言語標準ライブラリ関数である。この関数のプロトタイプを読み込むために 6 行目 の#include 指令を入れている。
- このソフトウェア部品提供の枠組みを利用した例を次に示す。

```
[motoki@x205a]$ cat -n useStackChar_ver2.cpp
     1 /* "StackChar_ver2.h"の利用例                                    */
```

```
 2 /* (1) 文字列を 1 個読み込みそれをスタックを用いて反転した後出力 */
 3 /* (2) デフォルトコンストラクタの利用                          */
 4 /* (3) コピーコンストラクタの利用                              */
 5
 6 #include <iostream>
 7 #include <string>
 8 #include "StackChar_ver2.h"
 9 using namespace std;
10
11 int main(void)
12 {
13   StackChar stackA(100);
14   string s;
15
16   cout << "Input a string:  ";
17   cin >> s;
18   for (int i=0; i < s.length(); ++i)
19     stackA.pushdown(s[i]);
20   for (int i=0; !stackA.isEmpty(); ++i)
21     s[i] = stackA.popup();
22   cout << "Reversed string: " << s << endl;
23   cout << "---" << endl;
24
25   StackChar stackB;       //デフォルトコンストラクタの利用
26   StackChar stack[3];     //デフォルトコンストラクタの利用
27   for (int i=0; i<3; ++i) {
28     for (int k=0; k<=i; ++k)
29       stack[i].pushdown('*');
30   }
31   stackB.showContents();
32   for (int i=0; i<3; ++i)
33     stack[i].showContents();
34   cout << "---" << endl;
35
36   cout << "s = \"" << s << "\"" << endl;
37   StackChar stackC(s);
38   StackChar stackD(stackC);  //コピーコンストラクタの利用
39   stackD.pushdown('*');
40   StackChar stackE = stackD; //コピーコンストラクタの利用？
41   stackE.popup();
42   stackE.pushdown('#');
43   stackC.showContents();
44   stackD.showContents();
```

```
    45   stackE.showContents();
    46   cout << "---" << endl;
    47
    48   cout << "生成されたStackCharインスタンス数 = "
    49        << StackChar::getNumOfInstances() << endl;
    50 }
[motoki@x205a]$ g++ useStackChar_ver2.cpp
[motoki@x205a]$ ./a.out
Input a string:  abc123
Reversed string: 321cba
---
stack_of_char(id=1,size=100) has 0 elements as follows:
  [Bottom] <--
stack_of_char(id=2,size=100) has 1 elements as follows:
  [Bottom] *  <--
stack_of_char(id=3,size=100) has 2 elements as follows:
  [Bottom] *  *  <--
stack_of_char(id=4,size=100) has 3 elements as follows:
  [Bottom] *  *  *  <--
---
s = "321cba"
stack_of_char(id=5,size=6) has 6 elements as follows:
  [Bottom] 3  2  1  c  b  a  <--
stack_of_char(id=6,size=16) has 7 elements as follows:
  [Bottom] 3  2  1  c  b  a  *  <--
stack_of_char(id=7,size=16) has 7 elements as follows:
  [Bottom] 3  2  1  c  b  a  #  <--
---
生成されたStackCharインスタンス数 = 8
[motoki@x205a]$
```

ここで、

◇ 利用例のプログラム 25~26 行目 の変数宣言、配列宣言ではコンストラクタが明示されていないので、StackChar_ver2.h の 15 行目,38~42 行目で定義されたデフォルトコンストラクタが暗黙に用いられる。

◇ 利用例のプログラム 40 行目 において、もし stackE という名前で一旦構成された StackChar インスタンスに変数 stackD の保持している内容が代入されているのであれば、const メンバ id への代入を行おうとしてエラーになったり、メンバ element の保持するポインタ値がコピーされ 2 つのインスタンス stackD と stackE が配列領域を共有したりするはずである。しかし、実行結果をみるとそういうことになっていないので、40 行目 でも、コピーコンストラクタ StackChar(stackD) が直接 stackE インスタンスの初期化を行なっているものと判断できる。

◇ 利用例のプログラム 49 行目 に見られる様に、静的メンバ関数を利用する時はメンバアクセス演算子 ”.” ではなくスコープ解決演算子 ”::” を用いる。

3.4 ソースファイルの構成

{柴田 (1999)2.1.4 節, 柴田 (2009)10.2 節,9.3 節,11.2 節,
Stroustrup(2015, 第 4 版)2.4.1 節,
Stroustrup(2015, エッセンス)3.2 節}

クラスの仕様部と実装部の分離： 定義したクラスは (自分または他人が) 将来再利用する可能性もある。ただ、再利用の際は、多くの場合、クラスを構成するメンバの情報やクラスで公開されている関数の仕様／使い方が分かれば十分で、出来上がっているクラスの実装部 (e.g. メンバ関数の本体部) を見る必要がない。そこで、
クラスを定義するファイルを構成する際は、

- クラスの仕様部と実装部をソースファイル上で分離すべきである。具体的には、
- クラスの構成メンバを記述した

```
class [クラス名] {
        (メンバの記述)
};
```

という部分 1 個と、inline 宣言するメンバ関数の定義、およびこれらに関連した記述だけから成る [クラス名].h という名前のヘッダファイルを構成する。その際、

◇ 将来このクラスを利用する時のために、十分なコメントを入れておく。

◇ 将来このクラスが色々な所で使われる様になると、① [クラス名].h を#include したヘッダファイルと② [クラス名].h の両方を#include するという事態も起こり得る。この様に[クラス名].h が重ねて#include された場合でも、「重複定義」というコンパイルエラーが出ない様にしなければならない。そのために、**インクルードガード**という定石手法に従って、例えば次の様に全体を #if と #endif で囲んで [クラス名].h を構成する。

```
#ifndef ___Class_[クラス名]
#define ___Class_[クラス名]
        (クラス定義等)
#endif
```

- 上記のヘッダファイル [クラス名].h に含まれない

◇ メンバ関数の詳細な定義 (**注意**：inline 関数は[クラス名].h の方に入れる)、

◇ 静的メンバ変数への初期設定、

◇ #include [クラス名].h

等の記述から成る [クラス名].cpp という名前のファイルを構成する。そして、将来の保守のために、コメントを入れておく。

そして、この様なファイル構成のクラスを利用する際は、

- クラス利用のソースファイル上で、#include [クラス名].h とする。
- ソースファイル [クラス名].cpp も指定してコンパイルする。

実装例 3.7 (char 型データが入るスタック, 部品提供の実装案 7) 実装例 3.6 で与えたク

ラス定義のファイル StackChar_ver2.h を上記の考えに従って仕様部 StackChar.h と実装部 StackChar.cpp の 2 つに分離した例を次に示す。

```
[motoki@x205a]$ cat -n StackChar.h
     1 /* 「char 型データが入るスタック」オブジェクトのクラス */
     2 /*  StackChar の仕様部                                  */
     3
     4 #ifndef ___Class_StackChar
     5 #define ___Class_StackChar
     6
     7 #include <string>
     8
     9 class StackChar {
    10   static int numOfInstances;//これまでに生成した StackChar インスタンスの個数
    11   const int id;     // インスタンスに固有の id 番号
    12   char* element;    // スタック領域へのポインタ
    13   int  size;        // スタックの容量
    14   int  top;         // スタックの top 要素を保持する配列要素の番号
    15 public:
    16   explicit StackChar(int size = 100); // スタック容量=size として初期化。
    17                                       // (デフォルトコンストラクタの役割も)
    18   StackChar(const StackChar& stack);  // コピーコンストラクタ。
    19   StackChar(const std::string& str);//引数の文字列をスタックに入れて初期化。
    20   StackChar(int size,                // 容量が size のスタック領域を確保し、
    21             const std::string& str); //  そこに引数の文字列を入れて初期化。
    22   ~StackChar() { delete[] element; }
    23   // オブジェクト (もしくは StackChar クラス全体) の情報を提供するための関数群
    24   static int getNumOfInstances() { return numOfInstances; }
    25                     // これまでに生成した StackChar インスタンスの個数を返す。
    26   int getId() const { return id; }            // スタックの id 番号を返す。
    27   int getSize() const { return size; }           // スタック容量を返す。
    28   int getNumOfElements() const { return top+1; }//スタック内の要素数を返す
    29   void showContents() const;                     // スタックの状況を出力。
    30   // StackChar 型オブジェクトを操作するための関数群
    31   void reset() { top = -1; }                  // スタックを空にする。
    32   void resize(int size);                      // スタック容量を変更。
    33   bool isEmpty() const { return (top < 0); }  // スタックが空か否か。
    34   void pushdown(char c);                      // pushdown 操作。
    35   char popup();                               // popup 操作。
    36 };
    37
    38 #endif
[motoki@x205a]$ cat -n StackChar.cpp
     1 /* 「char 型データが入るスタック」オブジェクトのクラス */
     2 /*  StackChar の実装部                                  */
     3
     4 #include <iostream>
     5 #include <string>
     6 #include <cstdlib>    // for exit()
     7 #include <cstring>    // for memcpy()
     8 #include "StackChar.h"
     9 using namespace std;
    10
```

```
11 // static変数の初期化 ----------------------------------------
12 int StackChar::numOfInstances = 0;
13
14 // 各種コンストラクタ ----------------------------------------
15 StackChar::StackChar(int size)      // スタック容量=sizeとして初期化
16                                     // (デフォルトコンストラクタの役割も)
17           : id(numOfInstances++), size(size), top(-1)
18 {
19   element = new char[size];
20 }
21
22 StackChar::StackChar(const StackChar& stack)     // コピーコンストラクタ
23           : id(numOfInstances++), size(stack.size), top(stack.top)
24 {
25   element = new char[stack.size];
26   memcpy(element, stack.element, stack.size);
27 }
28
29 StackChar::StackChar(const string& str) //引数文字列をスタックに入れて初期化
30           : id(numOfInstances++), size(str.length()), top(str.length()-1)
31 {
32   element = new char[size];
33   for (int i=0; i<str.length(); i++)
34     element[i] = str[i];
35 }
36
37 StackChar::StackChar(int size,       //容量がsizeのスタック領域を確保し、
38                      const string& str)//そこに引数の文字列を入れて初期化
39           : id(numOfInstances++), size(size), top(str.length()-1)
40 {
41   element = new char[size];
42   for (int i=0; i<str.length(); i++)
43     element[i] = str[i];
44 }
45
46 // オブジェクトの情報を提供するための関数群 ----------------------------
47 void StackChar::showContents() const              // スタックの内容を表示
48 {
49   cout << "stack_of_char(id=" << id << ",size=" << size << ") has "
50        << top+1 << " elements as follows:" << endl;
51   cout << "  [Bottom] ";
52   for (int i=0; i<=top; i++)
53     cout << element[i] << "  ";
54   cout << "<--" << endl;
55 }
56
57 // StackChar型オブジェクトを操作するための関数群 ----------------------
58 void StackChar::resize(int size)                  // スタック容量を変更
59 {
60   if (top >= size) {
61     cout << "insufficient stack size" << endl;
62     exit(1);
```

```
    63   }
    64   char* temp = new char[size];
    65   memcpy(temp, element, top+1);
    66   delete[] element;
    67   this->size = size;
    68   element = temp;
    69 }
    70
    71 void StackChar::pushdown(char c)                    // pushdown 操作
    72 {
    73   if (++top >= size) {
    74     this->resize(size+10);
    75   }
    76   element[top] = c;
    77 }
    78
    79 char StackChar::popup()                              // popup 操作
    80 {
    81   if (top < 0) {
    82     cout << "popup from empty stack" << endl;
    83     exit(1);
    84   }
    85   return element[top--];
    86 }
[motoki@x205a]$
```

これに関して、

- 実装部のソースコードはもはや他からインクルードされることもないので、見易さのために StackChar.cpp の 9 行目で using namespace std; としている。
- このソフトウェア部品提供の枠組みを実装例 3.6 の場合に倣って利用した例を次に示す。

```
[motoki@x205a]$ cat -n useStackChar.cpp
     1 /* "StackChar_ver3.h, StackChar_ver3.cpp"の利用例                */
     2 /* (1) 文字列を 1 個読み込みそれをスタックを用いて反転した後出力 */
     3 /* (2) デフォルトコンストラクタの利用                             */
     4 /* (3) コピーコンストラクタの利用                                 */
     5
     6 #include <iostream>
     7 #include <string>
     8 #include "StackChar.h"
     9 using namespace std;
    10
    11 int main(void)
    12 {
    13   StackChar stackA(100);
    14   string s;
    15
    16   cout << "Input a string:  ";
    17   cin >> s;
    18   for (int i=0; i < s.length(); ++i)
    19     stackA.pushdown(s[i]);
    20   for (int i=0; !stackA.isEmpty(); ++i)
```

```
    21     s[i] = stackA.popup();
    22   cout << "Reversed string: " << s << endl;
    23   cout << "---" << endl;
    24
    25   StackChar stackB;       //デフォルトコンストラクタの利用
    26   StackChar stack[3];     //デフォルトコンストラクタの利用
    27   for (int i=0; i<3; ++i) {
    28     for (int k=0; k<=i; ++k)
    29       stack[i].pushdown('*');
    30   }
    31   stackB.showContents();
    32   for (int i=0; i<3; ++i)
    33     stack[i].showContents();
    34   cout << "---" << endl;
    35
    36   cout << "s = ¨ << s << "¨ << endl;
    37   StackChar stackC(s);
    38   StackChar stackD(stackC);  //コピーコンストラクタの利用
    39   stackD.pushdown('*');
    40   StackChar stackE = stackD; //コピーコンストラクタの利用？
    41   stackE.popup();
    42   stackE.pushdown('#');
    43   stackC.showContents();
    44   stackD.showContents();
    45   stackE.showContents();
    46   cout << "---" << endl;
    47
    48   cout << "生成された StackChar インスタンス数 = "
    49        << StackChar::getNumOfInstances() << endl;
    50 }
[motoki@x205a]$ g++ useStackChar.cpp StackChar.cpp
[motoki@x205a]$ ./a.out
Input a string:  abc123
Reversed string: 321cba
---
stack_of_char(id=1,size=100) has 0 elements as follows:
  [Bottom] <--
stack_of_char(id=2,size=100) has 1 elements as follows:
  [Bottom] *  <--
stack_of_char(id=3,size=100) has 2 elements as follows:
  [Bottom] *  *  <--
stack_of_char(id=4,size=100) has 3 elements as follows:
  [Bottom] *  *  *  <--
---
s = "321cba"
stack_of_char(id=5,size=6) has 6 elements as follows:
  [Bottom] 3  2  1  c  b  a  <--
stack_of_char(id=6,size=16) has 7 elements as follows:
  [Bottom] 3  2  1  c  b  a  *  <--
stack_of_char(id=7,size=16) has 7 elements as follows:
  [Bottom] 3  2  1  c  b  a  #  <--
---
```

```
生成された StackChar インスタンス数 = 8
[motoki@x205a]$
```

3.5 文法的な諸注意 (まとめ)

{Pohl(1999)4.1 節,4.3-5 節,4.8 節,5 章前書き,5.1-3 節,5.6 節 }

class 宣言,struct 宣言の大枠： カプセル化されたソフトウェア部品 (オブジェクト) を作り出すために、C++言語では、オブジェクトの枠組み／設計図を定めた次の形の記述を行う。

```
class [クラス名] {
    (メンバの記述)
};
```

もしくは、

```
struct [クラス名] {
    (メンバの記述)
};
```

ここで、

- 「メンバの記述」の部分は、作り出すオブジェクトの構成メンバ (e.g. データ, 関数) の記述を並べたものである。(C 言語構造体の書式の拡張。)
- **メンバの公開／非公開に関しては、**
  - ◇ キーワードが struct の場合 はメンバは原則**公開**でオブジェクト外からのアクセスは暗黙に許可される。一方、キーワードが class の場合 はメンバは原則**非公開**でオブジェクト外からのアクセスは暗黙に禁止される。
  - ◇ 「メンバの記述」部に private: という宣言があれば、これ以降 (別の指示があるまで) のメンバは非公開になる。また、public: という宣言があれば、これ以降 (別の指示があるまで) のメンバは公開される。
- **メンバの宣言に static 修飾子が付いていれば、**
  そのメンバはクラス全体で共有される**静的メンバ**として扱われ、クラス外では [クラス名]::[メンバ名] という名前で表す。
  - ◇ 静的メンバ変数の初期化 はクラス定義の中で行うことはできず、クラス定義の外で static 修飾子も付けずに
    [データ型] [クラス名]::[静的メンバ変数名] = [初期値を表す式];
    という形で行う。
- **オブジェクトの構成メンバが関数の場合、**
  メンバの記述を並べた {...} の中に関数プロトタイプだけを書いても良いし、処理本体部を含む関数定義を書いても良い。

◇ メンバ関数の定義全体をメンバリスト {...} の中に書いた場合、この関数については暗黙に inline 宣言されたものとして処理される。
◇ メンバ関数のプロトタイプだけをメンバリスト {...} の中に書いた場合、この関数の処理本体部を含む関数定義はメンバリスト {...} の外で行う。その際の関数名は [クラス名]::[メンバ関数名] という名前で表す。

コンストラクタ： クラス定義に際しては、オブジェクトの構成メンバとして**コンストラクタ**と呼ばれる特殊な役割の関数を用意することができる。

- コンストラクタは、クラス定義に基づいたオブジェクトを生成する際に初期化のために自動的に呼び出される。
- コンストラクタの関数名は [クラス名] である。
- 戻り値も常に無いので、関数定義の際に「戻り値の型」の部分には何も書かない。
- オブジェクトの初期化に際して、オブジェクト内のデータメンバの初期化は、関数頭部と関数処理本体を囲む波括弧 {...} の間に次の形の記述を配置することによって行うことができる。

  : [メンバ名 1]([初期値を表す式 1]), ..., [メンバ名 n]([初期値を表す式 n])

  但し、メンバが基本データ型でなく、何らかのクラス定義に従ったオブジェクトである場合には

  [メンバ名]([コンストラクタへの実引数列])

  という形で初期設定の指定を書く。
- 一般に、引数なしで呼び出されるコンストラクタは**デフォルトコンストラクタ**と呼ばれ、当該クラスに属するオブジェクトを保持する変数や配列の宣言の際に初期設定の指定がない場合に暗黙に呼び出される、という特別な役割を果たす。
- 引数が 1 個のコンストラクタはデフォルトでは「引数の型 → コンストラクタが扱うオブジェクトの型」という型変換に使われる。この型変換を止めたい場合は関数宣言の前に explicit 修飾子を付ける。
- 一般に、同クラスに属する別インスタンスをコピーすることによってオブジェクトの初期化を行うコンストラクタを**コピーコンストラクタ**と呼ぶ。

デストラクタ： クラス定義に際しては、オブジェクトの構成メンバとして**デストラクタ**と呼ばれる特殊な役割の関数を用意することができる。

- デストラクタは、ブロック終了等に伴うオブジェクト消滅の際に後始末 (e.g. ヒープ領域から確保した領域の開放) のために自動的に呼び出される。
- デストラクタの関数名は ~[クラス名] である。
- 戻り値も常に無いので、関数定義の際に「戻り値の型」の部分には何も書かない。

演習問題

□演習 3.1 (e の 1000 桁計算) 前ターム「プログラミング AI」例題 8.4 で示した modules-napier-sub2.c に相当するオブジェクトをインスタンスとして生成するクラスを定義し、

これを利用して例題 8.4 で示された C プログラムと (ほぼ) 同等の処理を行う C++プログラムを構成せよ。

□**演習 3.2 (連想計算;Fibonacci 数列の再帰計算を効率的にする)** 前ターム「プログラミング AI」例題 8.5 で示した `modules-fibonacci-sub2.c` に相当するオブジェクトをインスタンスとして生成するクラスを定義し、これを利用して例題 8.5 で示された C プログラムと (ほぼ) 同等の処理を行う C++プログラムを構成せよ。

□**演習 3.3 (疑似乱数発生)** 前ターム「プログラミング AI」例題 8.6 で示した `modules-random.c` に相当するオブジェクトをインスタンスとして生成するクラスを定義し、これを利用して例題 8.6 で示された C プログラムと (ほぼ) 同等の処理を行う C++プログラムを構成せよ。

□**演習 3.4 (指定されたクラスの定義)** 漸化式 $a_{n+1} = (12345 * a_n + 17)\%100$ によって定まる数列 $a_0, a_1, a_2, a_3, ...$ を考える。これに関して、

(1) 次の 3 つのうち②以外を内部に持つオブジェクトをインスタンスとして生成する能力を備え、また②の様なコンストラクタを持つクラス `IntGenerator` を定義せよ。
   ① データメンバ `an` ··· 数列内で現在注目している要素の値を保持。
   ② コンストラクタ ··· データメンバ `an` に数列の第 0 項の値 a_0 を初期設定する。
   ③ メンバ関数 `next()` ··· データメンバ `an` 内に現在保持している値に続く、次の項の値を新しい `an` の値として設定し、これを戻り値として返す。

(2) 問 (1) で定義したクラス `IntGenerator` を利用して、$a_0 = 1$ を第 0 項とする数列 $a_0, a_1, a_2, a_3, ...$ の第 1 項〜第 100 項の値を出力する C++プログラムを作成せよ。(a_0 を出力から除外していることに注意。)

□**演習 3.5 (指定されたクラスの定義)** 次の (1), (2) の順に C++プログラムを構成せよ。

(1) 次の 5 つのうち③以外を内部に持つオブジェクトをインスタンスとして生成する能力を備え、また③の様なコンストラクタを持つクラス `Accumulator` を定義せよ。
   ① データメンバ `sum` ··· これまでに出てきた実数値データの総和を保持。
   ② データメンバ `num` ··· これまでに出てきた実数値データの個数を保持。
   ③ コンストラクタ ··· データメンバ `sum` を 0.0 に、`num` を 0 に初期設定する。
   ④ メンバ関数 `addNewData()` ··· 引数で指定された実数データを `sum` に加算し、`num` に 1 を加える。
   ⑤ メンバ関数 `getAverage()` ··· 現時点での $\frac{\texttt{sum}}{\texttt{num}}$ の値を求め、これを戻り値として返す。

(2) 問 (1) で定義したクラス `Accumulator` を利用して、$\frac{1}{10}\sum_{x=1}^{10}\frac{1}{x}$ を計算して出力する C++プログラムを作成せよ。

□**演習 3.6 (クラスの利用)** 次の C++プログラムを実行するとどういう出力が得られるか？下の会話の様子中で [　　] の部分に予想される出力文字列を入れよ。但し、解答の際は空白を ␣ と明示せよ。下の会話の様子では、下線部はキーボードからの入力を表している。

```
[motoki@x205a]$ cat Q.h
#ifndef ___Class_Q
#define ___Class_Q

#include <string>

class Q {
  int a, b, c;
public:
  Q(int a=0, int b=1, int c=2): a(a), b(b), c(c) {}
  int push(int x);
  int push(int x, int y);
  int push(int x, int y, int z);
  std::string config();
};

#endif
[motoki@x205a]$ cat Q.cpp
#include <sstream>
#include <string>
#include "Q.h"

using namespace std;

int Q::push(int x)
{
  int ans = a;
  a = b;
  b = c;
  c = x;
  return ans;
}

int Q::push(int x, int y)
{
  push(x);
  return push(y);
}

int Q::push(int x, int y, int z)
{
  push(x);
  push(y);
  return push(z);
}
```

(右上へ続く ↗)

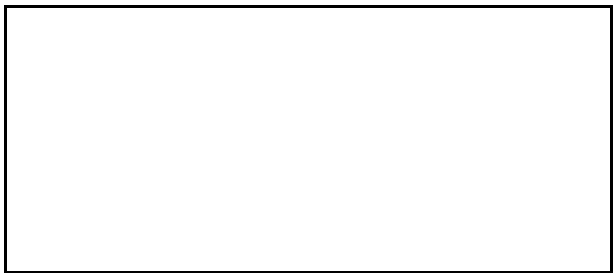
(↗ 左下からの続き。)

```
string Q::config()
{
  ostringstream os;
  os << "(a,b,c) = ("
     << a << "," << b << "," << c << ")";
  return os.str();
}
[motoki@x205a]$ cat test1808_2b.cpp
#include <iostream>
#include "Q.h"

using namespace std;

int main()
{
  Q obj1;
  cout << "(4)" << obj1.config() << endl;

  Q obj2(11, 22, 33);
  cout << "(5)" << obj2.config() << endl;
  cout << "(6)" << obj2.push(44) << endl;
  cout << "(7)" << obj2.push(55, 66) << endl;
  cout << "(8)" << obj2.push(77, 88, 99) << endl;
}
[motoki@x205a]$ g++ test1808_2b.cpp Q.cpp
[motoki@x205a]$ ./a.out
```



```
[motoki@x205a]$
```

4　何をどうクラスとして定義すべきか？

- クラス設計の基本方針
- クラス設計の例 (平面上の点, 長方形, 文字列, 複素数, 線形連結リスト, トランプ札の配り手)

4.1　クラス設計の基本方針

{プログラミング I(2017)19.5 節, Pohl(1999)4 章の前書き,4.3 節の最後}

何をクラスとして定義すべきか：　オブジェクト指向プログラミングでまず考えなければならないことは「どういうクラスを用意するか」である。これに関して、まず注意すべきことは、

- クラス定義の沿って作られるオブジェクトは、C 言語構造体のカプセルに操作方法も入れて機能拡充を図ったものである。それゆえ、元々、
- **オブジェクトの中心に位置するのは**、操作方法ではなく、(何らかのモノを表すために) 構造体内に**集められたデータ群**である、

ということである。従って、プログラム作りの課題があった時、一般的には、(課題で要求されている処理の記述に取り掛かる前に) まず、

① **課題文中の名詞もしくはそれに関連した物／エージェントの中から** 適切なもの を選び、

> 補足：ここでの「適切なもの (名詞)」としては、
> ◇ 汎用性のあるオブジェクトに繋がりそうなもの、
> ◇ 特殊だけれどもオブジェクト化するとプログラム全体が互いに独立な部分に綺麗に分割されそうなもの、など

② **その名詞もしくはその関連物／エージェントに相当するモノ 1 個を表すデータ群を定める**。そして、

③ 前ステップ②で定めたデータ群を操作するための関数群を定め、

④ 前ステップ②~③で定めたデータや関数のそれぞれについて公開／非公開を定める、

というクラス定義の作業を繰り返して、**問題領域に固有の抽象データ型** をクラス定義の形で必要なだけ実現する。

クラスを設計／定義する際の注意：

- 適切なクラス名、メンバ名を付ける。
  - ◇ クラス名としては 前段落ステップ① で選んだ「名詞やその関連物／エージェントに相当するモノ」を意味する**英単語 (列, 名詞句)** を採用すべき。
  - ◇ データメンバの名前としては その要素の役割に応じた**名詞 (句)** を採用すべき。
  - ◇ 関数の名前としては 行う操作を意味する**動詞 (句)** を採用すべき。
- クラス内に用意したメンバは**可能な限り非公開**とする。

> その恩恵：非公開メンバについては、内部の実装方法を自由に変更できる。(どう変更しても、クラスを利用している他のコードを変更する必要がない。)

- 目先の利用だけでなく、**将来の利用を想定**して、有用そうな操作は関数として用意しておく。

(補足) この講義ノートでは、Java 言語の習慣に倣って 2 つ目以降の単語の頭文字を大文字にして複数の単語を並べ、名前を構成している。しかし、C++言語では通常Stroustrup の書籍に倣って、単語と単語の間にアンダースコア (_, 下線記号) を入れて複数の単語を並べ、名前を構成する方式をとっている。

4.2 クラス設計の例

この節では、クラス設計の例を幾つか示す。

4.2.1 平面上の点のクラス

{Pohl(1999)4.5 節,5.1.4 節 }

例 4.1 (平面上の点のクラス) 平面上の点を扱う場合は、「平面上の点」を抽象的な 1 つのデータとして扱うことにより見通しの良いプログラムができると期待できる。そこで、例えば次の様に平面上の点のクラス Point2D を定義する。

```
[motoki@x205a]$ cat -n Point2D.h
     1 /* 「平面上の点」オブジェクトのクラス Point2D (仕様部) */
     2
     3 #ifndef ___Class_Point2D
     4 #define ___Class_Point2D
     5
     6 #include <string>
     7
     8 class Point2D {
     9   double x;
    10   double y;
    11 public:
    12   Point2D(double x = 0.0, double y = 0.0): x(x), y(y) {}
    13   //Point2D(const Point2D& pt)  //コピーコンストラクタ
    14   //  : x(pt.x), y(pt.y) {}     // (heap 領域からの領域確保がないので、
    15   //                            //  同等のものがデフォルトで用意される。)
    16   double getX() const { return x; }
    17   double getY() const { return y; }
    18   std::string toString() const; //内部保持の点座標を string データとして返す
    19   void setXY(double x = 0.0, double y = 0.0);  //点の座標を設定
    20   void plus(const Point2D& pt);                //点の座標を移動
    21   void plus(const double offsetX = 0.0, const double offsetY = 0.0);
    22 };
    23
    24 #endif
[motoki@x205a]$ cat -n Point2D.cpp
     1 /* 「平面上の点」オブジェクトのクラス Point2D (実装部) */
     2
     3 #include <sstream>
     4 #include <string>
     5 #include "Point2D.h"
     6 using namespace std;
```

```
 7
 8 // Point2D 型オブジェクトを操作するための関数群 ----------------------
 9 // オブジェクト内部に保持している点の座標を string 型データとして返す
10 string Point2D::toString() const
11 {
12   ostringstream os;
13   os << "(" << x << "," << y << ")";
14   return os.str();
15 }
16
17 // オブジェクト内部に保持している点の座標を設定
18 void Point2D::setXY(double x, double y)
19 {
20   this->x = x;
21   this->y = y;
22 }
23
24 // オブジェクト内部に保持している点の座標を移動
25 void Point2D::plus(const Point2D& pt)
26 {
27   x += pt.x;
28   y += pt.y;
29 }
30
31 void Point2D::plus(const double offsetX, const double offsetY)
32 {
33   x += offsetX;
34   y += offsetY;
35 }
[motoki@x205a]$
```

これに関して、

- C++では標準ライブラリ内に文字列への出力を行うためのストリーム (i.e. 出力用の**文字列ストリーム**) のクラス ostringstream が用意されている。
  - ◇ 上の Point2D.cpp の 3 行目 ではこれを使うためのヘッダファイルをインクルードしている。
  - ◇ そして、12 行目 では出力用の文字列ストリームを表す変数 os が確保され、
  - ◇ 続く 13 行目 でこの文字列ストリーム os に点の座標を表す文字列が流し込まれる。
  - ◇ 最後に 14 行目 では、ostringstream クラスに備わったメンバ関数 str() を用いて、文字列ストリーム os に蓄えられた文字列を string 型データとして取り出している。
- 上のクラス Point2D が定義されていれば、それを使って次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n usePoint2D.cpp
     1 /* Point2D.h, Point2D.cpp の利用例 */
     2
     3 #include <iostream>
     4 #include <string>
     5 #include "Point2D.h"
     6 using namespace std;
```

```
 7
 8 inline double square(const double x) { return x*x; }
 9
10 int main()
11 {
12   Point2D p1(0.5), p2(p1), q[11];
13
14   p2.plus(1,2);
15   for (int i=0; i<=10; ++i) {
16     double x = static_cast<double>(i) / 10.0;
17     q[i].setXY(x, square(x));
18   }
19
20   cout << "p1=" << p1.toString() << endl;
21   cout << "p2=" << p2.toString() << endl;
22   for (int i=0; i<=10; ++i) {
23     cout << "q[" << i << "]=" << q[i].toString() << endl;
24   }
25 }
[motoki@x205a]$ g++ usePoint2D.cpp Point2D.cpp
[motoki@x205a]$ ./a.out
p1=(0.5,0)
p2=(1.5,2)
q[0]=(0,0)
q[1]=(0.1,0.01)
q[2]=(0.2,0.04)
q[3]=(0.3,0.09)
q[4]=(0.4,0.16)
q[5]=(0.5,0.25)
q[6]=(0.6,0.36)
q[7]=(0.7,0.49)
q[8]=(0.8,0.64)
q[9]=(0.9,0.81)
q[10]=(1,1)
[motoki@x205a]$
```

ここで、

◇ この利用例のプログラム12行目の宣言により、Point2D型の変数p1,p2と配列要素q[0]～q[10]の領域がスタック領域に確保される。その際、p1(0.5)と書くことによりコンストラクタPoint2D(0.5)すなわちPoint2D(0.5,0.0)が呼び出されて変数p1の初期設定が行われ、またp2(p1)と書くことにより暗黙に定義されたコピーコンストラクタ(i.e.Point2D.hの10～11行目に相当するもの)が呼び出されて変数p2の初期設定が、q[11]と書くことによりデフォルトコンストラクタPoint2D()すなわちPoint2D(0.0,0.0)が呼び出されて配列要素q[0]～q[10]の初期設定が行われる。

4.2.2 長方形のクラス

{プログラミングI(2017)例題19.4}

例題 4.2 (長方形のクラス) 標準入力から 3 つの長方形のデータ (横幅と高さ) を読み取り、その中から最大面積の長方形の情報を出力する C++プログラムを作成せよ。

(考え方) 1 個の長方形の横幅と高さをバラバラに保持するのではなく 1 箇所に集めて操作方法と共に 1 つのカプセルの中に入れて抽象的な 1 つのデータとして扱うことにより、見通しが良く間違いが少ないプログラムを構築できると考えられる。そこで、プログラム中で 1 個の長方形を表すオブジェクトの型枠となるクラス Rectangle を定義することにする。その際、

- ソフトウェア部品としては汎用性のあるものが望ましいので、個々の Rectangle インスタンスには固有の id 番号を保持させることにし、id 番号の一意性を保証するために最新の id 番号 (=過去に生成した Rectangle インスタンスの個数) を常に保持する領域を Rectangle のメンバとして用意する。また、
- 十分に情報隠蔽も考慮すべきであるので、個々のデータメンバは外部から直接アクセスできない様にし、一方では外部からの正当な利用ができる様に参照や値変更の関数メンバを用意する。

(Rectangle クラスの定義) Rectangle クラスの定義例を次に示す。

```
[motoki@x205a]$ cat -n Rectangle.h
     1 /* 長方形オブジェクトのクラス Rectangle（仕様部）*/
     2
     3 #ifndef ___Class_Rectangle
     4 #define ___Class_Rectangle
     5
     6 #include <string>
     7
     8 class Rectangle {
     9   static int numOfInstances;  // これまでに生成したインスタンスの個数
    10   const int id;     // 長方形インスタンスに付ける id 番号
    11   double width;     // 長方形の横幅
    12   double height;    // 長方形の高さ
    13 public:
    14   Rectangle(double width = 0.0, double height = 0.0)
    15     : id(numOfInstances++), width(width), height(height) {}
    16   Rectangle(const Rectangle& rectangle);     //コピーコンストラクタ
    17   //~Rectangle() {}
    18   // オブジェクト (もしくは Rectangle クラス全体) の情報を提供するための関数群
    19   static int getNumOfInstances() { return numOfInstances; }
    20                       //これまでに生成した Rectangle インスタンスの個数を返す
    21   double getId() const { return id; }
    22   double getWidth() const { return width; }
    23   double getHeight() const { return height; }
    24   void setWidth(double width) { this->width = width; }
    25   void setHeight(double height) { this->height = height; }
    26   double getArea() const { return width*height; }  //長方形の面積を返す
    27   std::string toString() const; //内部の長方形情報を string データとして返す
    28 };
    29
    30 #endif
```

```
[motoki@x205a]$ cat -n Rectangle.cpp
     1 /* 長方形オブジェクトのクラス Rectangle (実装部) */
     2
     3 #include <sstream>
     4 #include <string>
     5 #include "Rectangle.h"
     6 using namespace std;
     7
     8 // static 変数の初期化 ------------------------------------------
     9 int Rectangle::numOfInstances = 0;
    10
    11 // 各種コンストラクタ --------------------------------------------
    12 Rectangle::Rectangle(const Rectangle& rectangle)   // コピーコンストラクタ
    13           : id(numOfInstances++),
    14     width(rectangle.width), height(rectangle.height) {}
    15
    16 // Rectangle 型オブジェクトを操作するための関数群 ----------------------
    17 // オブジェクト内部に保持している長方形情報を string 型データとして返す
    18 string Rectangle::toString() const
    19 {
    20   ostringstream os;
    21   os << "rectangle(id=" << id
    22      << ",width=" << width << ",height=" << height << ")";
    23   return os.str();
    24 }
[motoki@x205a]$
```

(Rectangle クラスの利用) Rectangle クラスが上の様に定義されていれば、それを利用して例題 4.2 で要求されている作業を例えば次の C++プログラムの様に表すことができる。

```
[motoki@x205a]$ cat -n findMaxAmong3Rectangles.cpp
     1 /* Rectangle.h, Rectangle.cpp の利用例                                  */
     2 /* 次の作業を順に行う C++プログラム                                     */
     3 /* (1) 標準入力から得られたデータを基に Rectangle インスタンスを 3 つ生成 */
     4 /* (2) 面積最大のインスタンスを見つけてその情報を出力                   */
     5
     6 #include <iostream>
     7 #include <string>
     8 #include "Rectangle.h"
     9 using namespace std;
    10
    11 int main()
    12 {
    13   Rectangle* rectangle[3];
    14
    15   //標準入力からデータを入力、それを基に長方形インスタンスを生成
    16   for (int i=0; i<3; ++i) {
    17     double width, height;
    18     cout << "長方形の幅と高さを入力 : ";
    19     cin >> width >>height;
    20     rectangle[i] = new Rectangle(width, height);
```

```
21   }
22
23   //生成された長方形インスタンスの内部情報を出力
24   cout << "生成された長方形インスタンス：" << endl;
25   for (int i=0; i<3; ++i) {
26     cout << "*rectangle[" << i << "]=" << rectangle[i]->toString() << endl;
27   }
28
29   //面積最大の長方形インスタンスを見つけてその情報を出力
30   int indexOfMaxArea = 0;
31   double maxArea     = rectangle[0]->getArea();
32   for (int i=1; i<3; ++i) {
33     double area = rectangle[i]->getArea();
34     if (area > maxArea) {
35       indexOfMaxArea = i;
36       maxArea        = area;
37     }
38   }
39   cout << "==>面積最大のものは"
40        << rectangle[indexOfMaxArea]->toString() << endl;
41
42   //heap 領域から確保したメモリを開放
43   for (int i=0; i<3; ++i)
44     delete rectangle[i];
45 }
[motoki@x205a]$ g++ findMaxAmong3Rectangles.cpp Rectangle.cpp
[motoki@x205a]$ ./a.out
長方形の幅と高さを入力：3  17
長方形の幅と高さを入力：10  10
長方形の幅と高さを入力：15  5
生成された長方形インスタンス：
*rectangle[0]=rectangle(id=0,width=3,height=17)
*rectangle[1]=rectangle(id=1,width=10,height=10)
*rectangle[2]=rectangle(id=2,width=15,height=5)
==>面積最大のものは rectangle(id=1,width=10,height=10)
[motoki@x205a]$
```

ここで、この利用例のプログラムでは、ブロック内の局所配列宣言によって Rectangle 型配列要素をスタック領域上に予め確保するのを止め、代わりに、個々の Rectangle インスタンスの初期設定値が決まった時点で初めてヒープ領域からメモリ確保をしてコンストラクタを呼び出す様にしている。そのため、

- この利用例のプログラム 13 行目 では、Rectangle インスタンスへのポインタを入れる配列が用意されているだけである。
- プログラム 20 行目 では、Rectangle インスタンスのための領域がヒープ領域から確保され、コンストラクタ Rectangle(width, height) によって初期化される。(領域確保と適切な初期化が同時に行われ分かり易い。)
- プログラム 43~44 行目 では、ブロック終了に伴って、ヒープ領域から確保したメモリの開放を行なっている。(このプログラムの場合は必要ないが、こういう習慣をつけておいた方が無難。)

4.2.3 文字列のクラス

{Pohl(1999)5.4 節, 柴田 (1999)3.4 節 }

例 4.3 (文字列のクラス) C++言語では文字列を扱い易くするために標準で string というクラスが用意されている。ここでは、この string クラスの様に、内部の配列容量を意識することなく任意の長さの文字列を表すための枠組みを自前で定義してみる。例えば次の通り。(クラス名 String)

```
[motoki@x205a]$ cat -n String.h
     1 /* 文字列オブジェクトのクラス String (仕様部) */
     2
     3 #ifndef ___Class_String
     4 #define ___Class_String
     5
     6 #include <iostream>
     7
     8 class String {
     9   char* s;
    10   int length;
    11 public:
    12   String();
    13   String(const char* stringWithNullTerminal); //変換コンストラクタ
    14   String(const String& string);               //コピーコンストラクタ
    15   ~String() { delete[] s; }
    16   // 文字列操作の関数群
    17   char charAt(int index) const;          //指定された添字番号の文字要素
    18   char& operator[](int index) const;
    19   void print() const { std::cout << "\"" << s << "\""; } //内部の文字列
    20                                                          //  情報を出力
    21   void println() const { std::cout << "\"" << s << "\"" << std::endl; }
    22   void assign(const char* stringWithNullTerminal); //代入
    23   void assign(const String& string);
    24   String& append(const char* stringWithNullTerminal); //最後尾に引数
    25                                                       //  文字列を連接
    26   String& append(const String& string);
    27   String& shrinkToSubstring(int from, int to); //部分文字列に縮小
    28                           //(引数のプラス値は index, マイナス値は削減幅と解釈)
    29 };
    30
    31 #endif
[motoki@x205a]$ cat -n String.cpp
     1 /* 文字列オブジェクトのクラス String (実装部) */
     2
     3 #include <iostream>
     4 #include <cstring>
     5 #include <cassert>
     6 #include "String.h"
     7 using namespace std;
     8
     9 // 各種コンストラクタ ------------------------------------------
    10 String::String(): length(0)
    11 {
    12   s = new char[1];
```

```
13   assert(s != 0);
14   s[0] = '\0';
15 }
16
17 String::String(const char* stringWithNullTerminal)
18 {
19   length = strlen(stringWithNullTerminal);
20   s = new char[length+1];
21   assert(s != 0);
22   strcpy(s, stringWithNullTerminal);
23 }
24
25 String::String(const String& string): length(string.length)
26 {
27   s = new char[length+1];
28   assert(s != 0);
29   strcpy(s, string.s);
30 }
31
32 // 文字列操作の関数群 ------------------------------------------
33 // 指定された添字番号の文字
34 char String::charAt(int index) const
35 {
36   assert(0 <= index && index < length);
37   return s[index];
38 }
39
40 char& String::operator[](int index) const
41 {
42   assert(0 <= index && index < length);
43   return s[index];
44 }
45
46 // 代入操作
47 void String::assign(const char* stringWithNullTerminal)
48 {
49   delete[] s;
50   length = strlen(stringWithNullTerminal);
51   s = new char[length+1];
52   assert(s != 0);
53   strcpy(s, stringWithNullTerminal);
54 }
55
56 void String::assign(const String& string)
57 {
58   if (this == &string)  // assignment of the form a = a;
59     return;
60   delete[] s;
61   length = string.length;
62   s = new char[length+1];
63   assert(s != 0);
64   strcpy(s, string.s);
```

```
 65 }
 66
 67 // 最後尾に引数文字列を連接
 68 String& String::append(const char* stringWithNullTerminal)
 69 {
 70   int initLength = length;
 71   length += strlen(stringWithNullTerminal);
 72   char* temp = new char[length+1];
 73   assert(temp != 0);
 74   strcpy(temp, s);
 75   strcpy(temp+initLength, stringWithNullTerminal);
 76   delete[] s;
 77   s = temp;
 78   return *this;
 79 }
 80
 81 String& String::append(const String& string)
 82 {
 83   int initLength = length;
 84   length += string.length;
 85   char* temp = new char[length+1];
 86   assert(temp != 0);
 87   strcpy(temp, s);
 88   strcpy(temp+initLength, string.s);
 89   delete[] s;
 90   s = temp;
 91   return *this;
 92 }
 93
 94 // 部分文字列に縮小 (引数のプラス値は index, マイナス値は削減幅と解釈)
 95 String& String::shrinkToSubstring(int from, int to)
 96 {
 97   assert(from<length && to <length);
 98   if (from < 0)           //マイナス値-->先頭からの削減幅と解釈
 99     from = - from;
100   if (to < 0)             //マイナス値-->最後尾からの削減幅と解釈
101     to = length -1 + to;
102   if (from <= to)
103     length = to - from + 1;
104   else
105     length = 0;
106   for (int i=0; i<length; ++i) //
107     s[i] = s[from+i];
108   s[length] = '\0';
109   return *this;
110 }
[motoki@x205a]$
```

これに関して、

- 上の String.h の 18 行目, String.cpp の 40∼44 行目 では、String 型変数の名前 *str* と添字番号 k を用いて添字 k の要素を *str*[k] という風に表すための演算子 [] を定義することを考えている。代入文の左辺にも置ける様にするために、戻り値の型は char

ではなく char& としている。

- String.cppの13行目, 21行目,28行目,36行目, 42行目,52行目,63行目, 73行目,86行目で用いている関数 assert() は、引数で与えられた条件が満たされないと強制終了を引き起こすC言語の標準ライブラリ関数である。
- 上のクラス String が定義されていれば、それを使って次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n useString.cpp
     1 /* String.h, String.cpp の利用例 */
     2
     3 #include <iostream>
     4 #include "String.h"
     5 using namespace std;
     6
     7 int main()
     8 {
     9   String x("abc"), y, z;
    10
    11   y.assign("123_");
    12   x[0] = 'A';
    13   for (int i=0; i<5; ++i)
    14     z.append(x).append(y);
    15   cout << "z=";
    16   z.println();
    17   z.shrinkToSubstring(2, -5);
    18   cout << "z.shrinkToSubstring(2, -5); " << endl << "==> z=";
    19   z.println();
    20 }
[motoki@x205a]$ g++ useString.cpp String.cpp
[motoki@x205a]$ ./a.out
z="Abc123_Abc123_Abc123_Abc123_Abc123_"
z.shrinkToSubstring(2, -5);
==> z="c123_Abc123_Abc123_Abc123_Ab"
[motoki@x205a]$
```

ここで、

◇ この利用例のプログラム9行目の宣言により、String型の変数x,y,zの領域がスタック領域に確保される。その際、xの初期設定にはString.cppの17〜23行目で定義されたコンストラクタが用いられ、y,zの初期設定にはString.cppの10〜15行目で定義されたデフォルトコンストラクタが用いられる。

4.2.4 複素数のクラス

{プログラミングI(2017)19.5節, Pohl(1999)4.5節, 柴田 (1999)3.3 節, 柴田 (2009)12.3 節, Stroustrup(2015, 第4版)3.2.1.1 節, Stroustrup(2015, エッセンス)4.2.1 節}

例題 4.4 (複素数のクラス) 非負整数データ n と複素数データ $C_n, C_{n-1}, ..., C_0$ を読み込み、これらの定数値の下で複素多項式 $C_n z^n + C_{n-1} z^{n-1} + C_{n-2} z^{n-2} + \cdots + C_1 z + C_0$ の値が

$$z = e^{i\pi k/5} = \cos\frac{\pi k}{5} + i \sin\frac{\pi k}{5} \qquad (k = 0, 1, 2, 3, ..., 9)$$

のそれぞれの値に対してどの様に変化するかを、表の形に見易く出力するC++プログラムを作成せよ。

(考え方) プログラム内で複素数をdouble型データと同じ様に扱うことが出来れば、複素多項式の計算も通常の実数値多項式を計算するコードとほぼ同じで済むことが予想される。そこで、ここでは複素数を抽象的な1つのデータとして扱うためのクラスComplexNumを定義することにする。その際、

- 将来の利用のために、複素数間の四則演算子だけでなく等価判定演算子 ==, != や複合代入演算子 +=, -=, *=, /= も使える様にする。
- 複素数とdouble型データの間の四則演算や等価判定演算も可能にしたいが、演算子の左側の要素がdouble型の場合の対応は、ComplexNumクラスの外で行わざるを得ない。そこで、一貫性を保つため、四則演算子と等価判定演算子の(多重)定義はComplexNumクラスの定義の外で行う。

(ComplexNumクラスの定義) ComplexNumクラスの定義例を次に示す。

```
[motoki@x205a]$ cat -n ComplexNum.h
     1 /* 複素数のクラス ComplexNum, ver.4 (仕様部) */
     2
     3 #ifndef ___Class_ComplexNum
     4 #define ___Class_ComplexNum
     5
     6 #include <iostream>
     7 #include <string>
     8
     9 class ComplexNum {
    10   double re;
    11   double im;
    12 public:
    13   ComplexNum(double re = 0.0, double im = 0.0): re(re), im(im) {}
    14   //ComplexNum(const ComplexNum& x) //コピーコンストラクタ
    15   //  : re(x.re), im(x.im) {}       // (heap領域からの領域確保がないので、
    16   //                                //  同等のものがデフォルトで用意される。)
    17   double getRe() const { return re; }
    18   double getIm() const { return im; }
    19   void setRe(double re) { this->re = re; }
    20   void setIm(double im) { this->im = im; }
    21   std::string toString() const;      // 保持している複素数をstring型で返す
    22   std::string toString(int precision) const;
    23                                      // 複合代入演算(下で定義)
    24   ComplexNum& operator+=(const ComplexNum& y);
    25   ComplexNum& operator-=(const ComplexNum& y);
    26   ComplexNum& operator*=(const ComplexNum& y);
    27   ComplexNum& operator/=(const ComplexNum& y);
```

```
28   ComplexNum& operator+=(double y);
29   ComplexNum& operator-=(double y);
30   ComplexNum& operator*=(double y);
31   ComplexNum& operator/=(double y);
32                                        // 四則演算 (メンバ関数外)
33   //friend ComplexNum operator+(ComplexNum x, const ComplexNum& y);
34   //friend ComplexNum operator-(ComplexNum x, const ComplexNum& y);
35   //friend ComplexNum operator*(ComplexNum x, const ComplexNum& y);
36   //friend ComplexNum operator/(ComplexNum x, const ComplexNum& y);
37   //friend ComplexNum operator+(ComplexNum x, double y);
38   //friend ComplexNum operator-(ComplexNum x, double y);
39   //friend ComplexNum operator*(ComplexNum x, double y);
40   //friend ComplexNum operator/(ComplexNum x, double y);
41   friend ComplexNum operator+(double x, const ComplexNum& y);
42   friend ComplexNum operator-(double x, const ComplexNum& y);
43   friend ComplexNum operator*(double x, const ComplexNum& y);
44   friend ComplexNum operator/(double x, const ComplexNum& y);
45   friend ComplexNum operator-(const ComplexNum& y);
46                                        // 等価判定演算 (メンバ関数外)
47   friend bool operator==(const ComplexNum& x, const ComplexNum& y);
48   friend bool operator!=(const ComplexNum& x, const ComplexNum& y);
49   friend bool operator==(const ComplexNum& x, double y);
50   friend bool operator!=(const ComplexNum& x, double y);
51   friend bool operator==(double x, const ComplexNum& y);
52   friend bool operator!=(double x, const ComplexNum& y);
53                                        // 入出力ストリームとの演算 <<, >>
54   friend std::ostream& operator<<(std::ostream& out, const ComplexNum& y);
55   friend std::istream& operator>>(std::istream& in, ComplexNum& y);
56 };
57
58 // inline 宣言する演算子の定義 ----------------------------------------
59 // (inline 宣言するメンバ関数についてはクラス定義と同じファイル内に入れる)
60
61 // 複合代入演算子 (ComplexNum += ComplexNum 等)
62 inline ComplexNum& ComplexNum::operator+=(const ComplexNum& y) {
63   re += y.re;
64   im += y.im;
65   return *this;
66 }
67
68 inline ComplexNum& ComplexNum::operator-=(const ComplexNum& y) {
69   re -= y.re;
70   im -= y.im;
71   return *this;
72 }
73
74 inline ComplexNum& ComplexNum::operator*=(const ComplexNum& y) {
75   double next_im = re*y.im+im*y.re;
76   re = re*y.re-im*y.im;
77   im = next_im;
78   return *this;
79 }
```

```
 80
 81 inline ComplexNum& ComplexNum::operator/=(const ComplexNum& y) {
 82   double next_im = (im*y.re-re*y.im)/(y.re*y.re+y.im*y.im);
 83   re = (re*y.re+im*y.im)/(y.re*y.re+y.im*y.im);
 84   im = next_im;
 85   return *this;
 86 }
 87
 88 // 複合代入演算子 (ComplexNum += double 等)
 89 inline ComplexNum& ComplexNum::operator+=(double y) {
 90   re += y;
 91   return *this;
 92 }
 93
 94 inline ComplexNum& ComplexNum::operator-=(double  y) {
 95   re -= y;
 96   return *this;
 97 }
 98
 99 inline ComplexNum& ComplexNum::operator*=(double  y) {
100   re *= y;
101   im *= y;
102   return *this;
103 }
104
105 inline ComplexNum& ComplexNum::operator/=(double  y) {
106   re /= y;
107   im /= y;
108   return *this;
109 }
110
111 // 四則演算子 (メンバ関数外, ComplexNum + ComplexNum 等)
112 inline ComplexNum operator+(ComplexNum x, const ComplexNum& y) { return x+=y; }
113 inline ComplexNum operator-(ComplexNum x, const ComplexNum& y) { return x-=y; }
114 inline ComplexNum operator*(ComplexNum x, const ComplexNum& y) { return x*=y; }
115 inline ComplexNum operator/(ComplexNum x, const ComplexNum& y) { return x/=y; }
116
117 // 四則演算子 (メンバ関数外, ComplexNum + double 等)
118 inline ComplexNum operator+(ComplexNum x, double y) { return x+=y; }
119 inline ComplexNum operator-(ComplexNum x, double y) { return x-=y; }
120 inline ComplexNum operator*(ComplexNum x, double y) { return x*=y; }
121 inline ComplexNum operator/(ComplexNum x, double y) { return x/=y; }
122
123 // 四則演算子 (メンバ関数外, double + ComplexNum 等)
124 inline ComplexNum operator+(double x, const ComplexNum& y)
125                     { return ComplexNum(x+y.re, y.im); }
126 inline ComplexNum operator-(double x, const ComplexNum& y)
127                     { return ComplexNum(x-y.re, -y.im); }
128 inline ComplexNum operator*(double x, const ComplexNum& y)
129                     { return ComplexNum(x*y.re, x*y.im); }
130 inline ComplexNum operator/(double x, const ComplexNum& y)
131                     { return ComplexNum(x*y.re/(y.re*y.re+y.im*y.im),
```

```
   132                                          -x*y.im/(y.re*y.re+y.im*y.im)); }
   133
   134 // 単項マイナス演算子（メンバ関数外, - ComplexNum)
   135 inline ComplexNum operator-(const ComplexNum& y)
   136                     { return ComplexNum(-y.re, -y.im); }
   137
   138 // 等価演算子（メンバ関数外, ComplexNum == ComplexNum 等)
   139 inline bool operator==(const ComplexNum& x, const ComplexNum& y)
   140                     { return (x.re==y.re && x.im==y.im); }
   141 inline bool operator!=(const ComplexNum& x, const ComplexNum& y)
   142                     { return (x.re!=y.re || x.im!=y.im); }
   143
   144 // 等価演算子（メンバ関数外, ComplexNum == double 等)
   145 inline bool operator==(const ComplexNum& x, double y)
   146                     { return (x.re==y && x.im==0.0); }
   147 inline bool operator!=(const ComplexNum& x, double y)
   148                     { return (x.re!=y || x.im!=0.0); }
   149
   150 // 等価演算子（メンバ関数外, double == ComplexNum 等)
   151 inline bool operator==(double x, const ComplexNum& y)
   152                     { return (x==y.re && 0.0==y.im); }
   153 inline bool operator!=(double x, const ComplexNum& y)
   154                     { return (x!=y.re || 0.0!=y.im); }
   155
   156 #endif
[motoki@x205a]$ cat -n ComplexNum.cpp
     1 /* 複素数のクラス ComplexNum, ver.4（実装部）*/
     2
     3 #include <sstream>
     4 #include <iostream>
     5 #include <iomanip>
     6 #include <string>
     7 #include "ComplexNum.h"
     8 using namespace std;
     9
    10 // 各種コンストラクタ ----------------------------------------
    11
    12 // 複素数オブジェクト操作の関数群 --------------------------------
    13 // 保持している複素数 string 型データとして返す
    14 string ComplexNum::toString() const {
    15   ostringstream ostr;
    16   ostr << "(" << re << ")+(" << im << ")i";
    17   return ostr.str();
    18 }
    19
    20 string ComplexNum::toString(int precision) const {
    21   ostringstream ostr;
    22   ostr << scientific << setprecision(precision)
    23        << "(" << setw(precision+7) << re << ")+("
    24        << setw(precision+7) << im << ")i";
    25   return ostr.str();
    26 }
```

```
    27
    28 // 入出力ストリームとの演算 ------------------------------
    29 // 出力ストリームへ ComplexNum 型データを挿入する演算子 << の多重定義
    30 ostream& operator<<(ostream& out, const ComplexNum& y) {
    31   return (out << y.toString());
    32 }
    33
    34 // 入力ストリームから ComplexNum 型データを抽出する演算子 >> の多重定義
    35 istream& operator>>(istream& in, ComplexNum& y) {
    36   return (in >> y.re >> y.im);
    37 }
[motoki@x205a]$
```

ここで、

- ComplexNum.h, 24〜55 行目の関数プロトタイプに現れる関数名は全て operator記号列 という形をしている。この種の関数は**演算子関数**と呼ばれるもので、定義することにより記号列 を演算子として使える様になる。
- ComplexNum.h の 24〜31 行目,62〜109 行目 で定義されている関数においては、演算子の右側の要素が引数 y として扱われ、複合代入演算子の左側の要素が演算実行のオブジェクトとして扱われる。
- ComplexNum.h の 33〜55 行目,112〜154 行目 で定義されている関数においては、演算子の右側の要素が第 2 引数 y として扱われ、演算子の左側の要素が第 1 引数 x (または out,in) として扱われる。
- ComplexNum.h の 41〜55 行目 に現れる friend は、本来なら private 宣言されて非公開になっている要素への参照を例外的に friend 宣言の右側の関数に許可することを示している。この様な関数を**フレンド関数**という。(getRe() や getIm() 経由でも参照できるが直接参照できた方が処理効率が良いので、頻繁に使う関数については friend 宣言は効果的である。)
- ComplexNum.h の 33〜40 行目 に現れる関数では処理の本体部で ComplexNum インスタンスの要素を参照しないので、friend 宣言も不要である。ただ、関連した関数としてどういうものが用意されているかを示すために、コメントアウトした上で並べている。

(ComplexNum クラスの利用) ComplexNum クラスが上の様に定義されていれば、それを利用して次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n useComplexNum.cpp
     1 /* ComplexNum.h, ComplexNum.cpp の利用例 */
     2
     3 #include <iostream>
     4 #include <string>
     5 #include "ComplexNum.h"
     6 using namespace std;
     7
     8 int main()
     9 {
    10   ComplexNum x, y, z;
    11
    12   cin >> x >> y;
```

```
    13   cout << "x=" << x << ", y=" << y << endl;
    14   z = (x+y+0.5)/5;
    15   cout << "==> z = (x+y+0.5)/5 = " << z << endl;
    16 }
[motoki@x205a]$ g++ useComplexNum.cpp ComplexNum.cpp
[motoki@x205a]$ ./a.out
1 2 3 4
x=(1)+(2)i, y=(3)+(4)i
==> z = (x+y+0.5)/5 = (0.9)+(1.2)i
[motoki@x205a]$
```

例題 4.4 で要求されている作業に関しては、例えば次の C++プログラムの様に表すことができる。(**注目**： 実数値多項式を計算するコードとほぼ同じで済んでいる。)

```
[motoki@x205a]$ cat -n calcComplexPolynomials.cpp
     1 /* 非負整数データ n と複素数データ C_n, C_n-1, ..., C_0 を読み込み、*/
     2 /* これらの定数値の下で複素多項式                                    */
     3 /*       C_n*z^n + C_n-1*z^n-1 + ... + C_1*z + C_0                */
     4 /* の値が                                                          */
     5 /*       z = exp(i π k/5)   (k=0,1,2, ..., 9)                     */
     6 /* のそれぞれの値に対してどの様に変化するかを、表の形に見易く出力する */
     7 /* C++プログラム                                                   */
     8
     9 #include <iostream>
    10 #include <iomanip>
    11 #include <cmath>
    12 #include "ComplexNum.h"
    13 using namespace std;
    14
    15 const int MAX_DEGREE = 100;
    16 const double PI = 3.1415926535897932;   // 円周率
    17
    18 int main()
    19 {
    20   int degree;
    21   ComplexNum coeff[MAX_DEGREE+1];
    22
    23   //多項式の次数と係数を標準入力から読み込む
    24   cout << "複素多項式の次数 (100 以下) : ";
    25   cin >> degree;
    26   for (int i=degree; i>=0; --i) {
    27     cout << i << "次係数の実部と虚部 : ";
    28     double re, im;
    29     cin >> re >> im;
    30     coeff[i].setRe(re);
    31     coeff[i].setIm(im);
    32   }
    33
    34   //入力された値の確認
    35   cout << "degree = " << degree << endl;
    36   for (int i=degree; i>=0; --i) {
    37     cout << "coeff[" << i << "] = " << coeff[i].toString() << endl;
    38   }
    39
```

```
    40   //多項式の値を計算して出力
    41   cout << " k                    z                "
    42        << "  c[d]*z^d+c[d-1]*z^(d-1)+ ... +c[1]*z+c[0]" << endl
    43        << "--  --------------------------------"
    44        << "  ----------------------------------------" << endl;
    45   for (int k=0; k<10; ++k) {
    46     ComplexNum z(cos(PI*k/5.0), sin(PI*k/5.0));
    47     ComplexNum result= coeff[degree];
    48     for (int i=degree-1; i>=0; --i)
    49       result = result*z + coeff[i];
    50     cout << setw(2) << k << "  "
    51          << z.toString(6) << "  "
    52          << result.toString(6) << endl;
    53   }
    54 }
[motoki@x205a]$ g++ calcComplexPolynomials.cpp ComplexNum.cpp
[motoki@x205a]$ ./a.out
複素多項式の次数 (100 以下)：3
3 次係数の実部と虚部：1.0 -2.0
2 次係数の実部と虚部：-3.0 4.0
1 次係数の実部と虚部：5.0 -6.0
0 次係数の実部と虚部：-7.0 8.0
degree = 3
coeff[3] = (1)+(-2)i
coeff[2] = (-3)+(4)i
coeff[1] = (5)+(-6)i
coeff[0] = (-7)+(8)i
 k                  z                  c[d]*z^d+c[d-1]*z^(d-1)+ ... +c[1]*z+c[0]
--  --------------------------------  ----------------------------------------
 0  ( 1.000000e+00)+( 0.000000e+00)i  (-4.000000e+00)+( 4.000000e+00)i
 1  ( 8.090170e-01)+( 5.877853e-01)i  (-2.566385e+00)+( 6.036813e+00)i
 2  ( 3.090170e-01)+( 9.510565e-01)i  (-1.657253e+00)+( 6.932006e+00)i
 3  (-3.090170e-01)+( 9.510565e-01)i  ( 1.572893e+00)+( 1.093085e+01)i
 4  (-8.090170e-01)+( 5.877853e-01)i  (-2.430068e+00)+( 2.021529e+01)i
 5  (-1.000000e+00)+( 1.224647e-16)i  (-1.600000e+01)+( 2.000000e+01)i
 6  (-8.090170e-01)+(-5.877853e-01)i  (-2.089617e+01)+( 6.728984e+00)i
 7  (-3.090170e-01)+(-9.510565e-01)i  (-1.219093e+01)+(-9.308531e-01)i
 8  ( 3.090170e-01)+(-9.510565e-01)i  (-6.016509e+00)+( 2.123722e+00)i
 9  ( 8.090170e-01)+(-5.877853e-01)i  (-5.815581e+00)+( 3.963187e+00)i
[motoki@x205a]$
```

4.2.5 線形連結リストのクラス

{プログラミング AI(2018)12.3 節, Pohl(1999)5.7 節, 柴田 (1999)3.5 節}

例題 4.5 (線形連結リストのクラス) 前ターム「プログラミング AI」の例題 12.3 では、

① 入力ストリームに現れる 識別子 (1 個以上の空白) 整数 という形のデータを読み込んでは、

② それを 識別子 に関して辞書順になる様に線形リストに登録する、

という作業を繰り返し、最後に線形リストに登録されたものを順に出力する C プログラムを作成した。これに対して、ここでは (ほぼ) 同等の処理を行うプログラムを C++ で構成してみよ。

(考え方) この例題の場合は、識別子と整数を要素にもつ多数の Node を識別子の辞書順に連結して保持するエージェントが居れば、全体の作業を見通し良く記述することができる。そこで、ここではこの種のエージェントをインスタンスとするクラス SortedLinkedListOfIdInt を定義することにする。その際、

- Node 内のデータ要素を非公開にすると SortedLinkedListOfIdInt エージェント内での作業に余分な手間がかかることが予想され、また Node 内に関数メンバを用意して協調動作させようとするとコードが分散されてしまうので、Node は公開のデータ要素とコンストラクタだけから成る構造体として表す。
- 上の例題 4.5 で必要な関数だけでなく、将来のため、あったら便利そうな関数をメンバ関数として用意する。

(SortedLinkedListOfIdInt クラスの定義) SortedLinkedListOfIdInt クラスの定義例を次に示す。

```
[motoki@x205a]$ cat -n SortedLinkedListOfIdInt.h
     1 /* 識別子と整数を要素にもつ Node 群を識別子の辞書順に連結した、*/
     2 /* 連結リストのクラス（仕様部）                                    */
     3
     4 #ifndef ___Class_SortedLinkedListOfIdInt
     5 #define ___Class_SortedLinkedListOfIdInt
     6
     7 #include <cstddef>   // for NULL
     8 #include <string>
     9
    10 struct NodeStringInt {
    11   const std::string id;
    12   const int num;
    13   NodeStringInt* next;
    14   NodeStringInt(std::string id="", int num=0, NodeStringInt* next=NULL)
    15     : id(id), num(num), next(next) {}
    16 };
    17
    18 class SortedLinkedListOfIdInt {
    19   NodeStringInt* head;
    20 public:
    21   SortedLinkedListOfIdInt(): head(NULL) {}
    22   ~SortedLinkedListOfIdInt() { removeAllNodes(); }
    23   NodeStringInt getHeadNodeInfo() const;          //先頭 Node 内の情報を返す
    24   NodeStringInt get1stNodeInfoOf(const std::string& id) const;
```

```
                                                  //引数指定の最初の Node 情報を返す
    25   int getNumOfNodesOf(const std::string& id) const;//引数指定の Node 数返却
    26   void printAllNodeInfo() const;               //全ての Node 内の情報を出力
    27   void insertNodeOf(const std::string& id, const int& num);//新データ挿入
    28   int removeHeadNode();                        //先頭 Node の削除
    29   int remove1stNodeOf(const std::string& id); //引数指定の最初の Node を削除
    30   int removeAllNodesOf(const std::string& id);//引数指定の全 Node を削除
    31   void removeAllNodes();                       //全 Node を削除
    32 };
    33
    34 #endif
[motoki@x205a]$ cat -n SortedLinkedListOfIdInt.cpp
     1 /* 識別子と整数を要素にもつ Node 群を識別子の辞書順に連結した、*/
     2 /* 連結リストのクラス（実装部）                                    */
     3
     4 #include <iostream>
     5 #include <iomanip>
     6 #include <cstddef>   // for NULL
     7 #include <string>
     8 #include "SortedLinkedListOfIdInt.h"
     9 using namespace std;
    10
    11 // 連結リストの先頭にある Node のコピーを返す
    12 NodeStringInt SortedLinkedListOfIdInt::getHeadNodeInfo() const {
    13   return NodeStringInt(head->id, head->num, NULL);
    14 }
    15
    16 // 連結リスト中で引数を id 要素とする最初の Node のコピーを返す
    17 NodeStringInt SortedLinkedListOfIdInt::
    18                         get1stNodeInfoOf(const string& id) const {
    19   NodeStringInt* ptr = head;
    20   while (ptr != NULL && ptr->id <= id) {
    21     if (ptr->id == id) {
    22       return NodeStringInt(ptr->id, ptr->num, NULL);
    23     }
    24     ptr = ptr->next;
    25   }
    26   return NodeStringInt("", 0, NULL);
    27 }
    28
    29 // 連結リスト中で引数を id 要素とする Node の個数を返す
    30 int SortedLinkedListOfIdInt::getNumOfNodesOf(const string& id) const {
    31   NodeStringInt* ptr = head;
    32   while (ptr != NULL && ptr->id < id)
    33     ptr = ptr->next;
    34   int count;
    35   for (count=0; ptr != NULL && ptr->id == id; ++count)
    36     ptr = ptr->next;
    37   return count;
    38 }
    39
    40 // 連結リスト内に保持されている内容を表の形に出力
```

```
41 void SortedLinkedListOfIdInt::printAllNodeInfo() const {
42   cout << "番号      num         id" << endl
43        << "----  ----------  ----------" << endl;
44   int count = 0;
45   for (NodeStringInt* ptr=head; ptr != NULL; ptr = ptr->next) {
46     cout << setw(4) << ++count
47          << setw(12) << ptr->num << "  " << ptr->id << endl;
48   }
49 }
50
51 // 引数要素をもつ Node を連結リスト内の id の辞書順を保つ位置に挿入
52 void SortedLinkedListOfIdInt::insertNodeOf(const string& id, const int& num) {
53   NodeStringInt** ptrptr = &head;
54   while ((*ptrptr)!=NULL && (*ptrptr)->id <= id)
55     ptrptr = &((*ptrptr)->next);
56   *ptrptr = new NodeStringInt(id, num, *ptrptr);
57 }
58
59 // 連結リスト中の先頭 Node を削除し、削除した Node 数を返す
60 int SortedLinkedListOfIdInt::removeHeadNode() {
61   if (head == NULL)
62     return 0;
63   NodeStringInt* tmp = head;
64   head = head->next;
65   delete tmp;
66   return 1;
67 }
68
69 // 連結リスト中で引数を id 要素とする最初の Node 削除し、削除した Node 数を返す
70 int SortedLinkedListOfIdInt::remove1stNodeOf(const string& id) {
71   NodeStringInt** ptrptr = &head;
72   while (*ptrptr != NULL && (*ptrptr)->id < id)
73     ptrptr = &((*ptrptr)->next);
74   if (*ptrptr != NULL && (*ptrptr)->id == id) {
75     NodeStringInt* tmp = *ptrptr;
76     *ptrptr = (*ptrptr)->next;
77     delete tmp;
78     return 1;
79   }else {
80     return 0;
81   }
82 }
83
84 // 連結リスト中で引数を id 要素とする Node を全て削除し、削除した Node 数を返す
85 int SortedLinkedListOfIdInt::removeAllNodesOf(const string& id) {
86   NodeStringInt** ptrptr = &head;
87   while (*ptrptr != NULL && (*ptrptr)->id < id)
88     ptrptr = &((*ptrptr)->next);
89   int count = 0;
90   for (count=0; *ptrptr != NULL && (*ptrptr)->id == id; ++count) {
91     NodeStringInt* tmp = *ptrptr;
92     *ptrptr = (*ptrptr)->next;
```

```
    93     delete tmp;
    94   }
    95   return count;
    96 }
    97
    98 // 連結リスト中の Node を全て削除
    99 void SortedLinkedListOfIdInt::removeAllNodes() {
   100   while (head != NULL) {
   101     NodeStringInt* tmp = head;
   102     head = head->next;
   103     delete tmp;
   104   }
   105 }
[motoki@x205a]$
```

(SortedLinkedListOfIdInt クラスの利用) SortedLinkedListOfIdInt クラスが上の様に定義されていれば、それを利用して例題 4.5 で要求されている作業を例えば次の C++ プログラムの様に表すことができる。

```
[motoki@x205a]$ cat -n readIdNumSeqAndPrintInLexicalOrder.cpp
     1 /* 線形連結リストを用いて識別子 (と整数) を辞書順に登録・出力 */
     2
     3 #include <iostream>
     4 #include <cstdlib>
     5 #include "SortedLinkedListOfIdInt.h"
     6 using namespace std;
     7
     8 int main()
     9 {
    10   SortedLinkedListOfIdInt list;
    11   string id;
    12   int num;
    13
    14   cin >> id >> num;
    15   while (cin.good()) {
    16     list.insertNodeOf(id, num);
    17     cin >> id >> num;
    18   }
    19
    20   if (! cin.eof()) {
    21     cout << "Input Error!" << endl;
    22     exit(EXIT_FAILURE);
    23   }
    24
    25   list.printAllNodeInfo();
    26 }
[motoki@x205a]$ g++ readIdNumSeqAndPrintInLexicalOrder.cpp SortedLinkedListOfIdInt.cpp
[motoki@x205a]$ ./a.out < dynamic-llist.data
番号      num         id
----  ----------  ----------
   1         123  abc
   2       55555  asdfghjk
```

```
   3          77  efghi
   4         456  kkk
   5        2345  ppp_qqq
   6      222222  qwertyui
   7         987  zxcv
[motoki@x205a]$
```

ここで、

- この利用例のプログラム 15 行目 中の good() は cin を始めとした istream オブジェクトに備わったメンバ関数で、入力時に何の問題も起きていない場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
- この利用例のプログラム 20 行目 中の eof() は cin を始めとした istream オブジェクトに備わったメンバ関数で、入力終端 (e.g. ファイルの終わり) に至っている場合は非ゼロの値を返し、それ以外の場合は 0 を返す。

4.2.6 トランプ札の配り手のクラス

{Pohl(1999)4.7 節 }

例題 4.6 (トランプ札の配り手のクラス) トランプの (Joker を除く) カード 5 枚がランダムに配られた時、それが Poker のストレートフラッシュになる確率、4 カードになる確率、フルハウスになる確率、フラッシュになる確率、ストレートになる確率、3 カードになる確率、2 ペアになる確率、1 ペアになる確率、ノーペアになる確率を **MonteCarlo 法**により見積もる C++プログラムを作成せよ。すなわち、コンピュータ上で①ランダムに 5 枚のカードを配り②その役を調べる、という作業を繰り返すシミュレーションを行い、それぞれの役が出る割合を求める C++プログラムを作成せよ。

(考え方) この例題の場合は、トランプの (Joker を除く) カードを公正にシャッフルして配る役割を担ったエージェントが居れば、あとは配られた手配の役を見極めることだけが主な作業となる。そこで、ここではこの種のエージェントをインスタンスとするクラス CardDealer を定義することにする。その際、

- 個々のカードの内容については、トランプゲームを行う際の基本データであり、非公開にすると煩わしいことも多いと考えられる。それゆえ、ここでは個々のカードを単純にスーツ (e.g. スペード) とランク (2~10,J,Q,K,A) を公開メンバとする構造体 Card として表す。
- スーツ (e.g. スペード) とランク (2~10,J,Q,K,A) は、どちらも、数値データとは言い難いので、列挙型定数として表す。
- 上の例題 4.6 で必要な関数だけでなく、将来のため、あったら便利そうな関数をメンバ関数として用意する。

(CardDealer クラスの定義) CardDealer クラスの定義例を次に示す。

```
[motoki@x205a]$ cat -n CardDealer.h
     1 /*  トランプのディーラーのクラス CardDealer (仕様部) */
     2
     3 #ifndef ___Class_CardDealer
```

```
 4 #define ___Class_CardDealer
 5
 6 #include <string>
 7
 8 enum Suit { CLUB, DIAMOND, HEART, SPADE };
 9 enum Rank { ILLEGAL, RANK2=2, RANK3, RANK4, RANK5, RANK6,
10             RANK7, RANK8, RANK9, RANK10, JACK, QUEEN, KING, ACE };
11
12 struct Card {
13   Suit suit;
14   Rank rank;
15   std::string toString() const {
16     const std::string SymbolOfSuit[4] = {"CLB", "DIA","HRT","SPD"};
17     const std::string SymbolOfRank[15] = {"","","_2","_3", "_4","_5","_6","_7",
18                                           "_8","_9","10","_J","_Q","_K","_A"};
19     return SymbolOfSuit[suit]+SymbolOfRank[rank];
20   }
21 };
22
23 class CardDealer {
24   Card card[52];          //Joker を除く 52 枚のカードを保持
25   int numOfDealedCards;   //配付済みで手元にないカードの枚数
26 public:
27   CardDealer();
28   void renewCardDeck();
29   void shuffle();
30   Card deal1Card();              //カードを 1 枚配る (戻り値が次のカード)
31   void dealCards(const int numOfCards, Card* hand);
32                   //引数で指定された枚数だけ、カードを引数指定の配列上に配る
33 };
34
35 #endif
[motoki@x205a]$ cat -n CardDealer.cpp
 1 /*  トランプのディーラーのクラス CardDealer (実装部) */
 2
 3 #include <iostream>
 4 #include <string>
 5 #include <cstdlib>
 6 #include <ctime>
 7 #include "CardDealer.h"
 8 using namespace std;
 9
10 Suit toSuit[4] = {CLUB, DIAMOND, HEART, SPADE};
11 Rank toRank[15] = {ILLEGAL, ILLEGAL, RANK2, RANK3, RANK4, RANK5, RANK6,
12                    RANK7, RANK8, RANK9, RANK10, JACK, QUEEN, KING, ACE};
13
14 // コンストラクタ ----------------------------------------
15 CardDealer::CardDealer()
16 {
17   renewCardDeck();
18   srand(time(NULL));
19   shuffle();
```

```
20 }
21
22 // カードデッキ操作の関数群 ------------------------------------------
23 // カードの束を新しいものと取り替える
24 void CardDealer::renewCardDeck()
25 {
26   int k = 0;
27   for (int suit=CLUB; suit<=SPADE; ++suit) {
28     card[k].suit = toSuit[suit];
29     card[k].rank = ACE;
30     ++k;
31     for (int rank=RANK2; rank<=KING; ++rank) {
32       card[k].suit = toSuit[suit];
33       card[k].rank = toRank[rank];
34       ++k;
35     }
36   }
37   numOfDealedCards = 0;
38 }
39
40 // 保持しているカードの束を混ぜる
41 void CardDealer::shuffle()
42 {
43   for (int k=numOfDealedCards; k<52; ++k) {
44     int i = k + (rand() % (52-k));
45     Card tmp = card[k];  //swap cards
46     card[k] = card[i];
47     card[i] = tmp;
48   }
49 }
50
51 // 次の 1 枚を配る
52 Card CardDealer::deal1Card()
53 {
54   if (numOfDealedCards >= 52) {
55     numOfDealedCards = 0;   //新しいカードを一組用意 (renewCardDeck() 省略)
56     shuffle();
57   }
58   return card[numOfDealedCards++];
59 }
60
61 // 引数指定の枚数を引数の配列上に配る
62 void CardDealer::dealCards(const int numOfCards, Card* hand)
63 {
64   if (numOfCards < 1 || 52 < numOfCards) {
65     cout << "Invalid number of cards is specified in dealCards()."
66          << endl;
67     exit(EXIT_FAILURE);
68   }else if (numOfCards > 52-numOfDealedCards) {
69     numOfDealedCards = 0;   //新しいカードを一組用意 (renewCardDeck() 省略)
70     shuffle();
71   }
```

```
    72   for (int k=0; k<numOfCards; ++k) {
    73     hand[k] = card[numOfDealedCards+k];
    74   }
    75   numOfDealedCards += numOfCards;
    76 }
[motoki@x205a]$
```

ここで、

- CardDealer.h の 13∼14 行目 に見られる様に、C++言語では 構造体タグと同じ様に **列挙タグ**もデータ型名として用いることができる。
- CardDealer.h の 15∼20 行目 では、各々の Card 型構造体の文字列表記を可能にするためのメンバ関数を定義している。(これによって、どういう文字列表記をするかは Card 型構造体自身に尋ねることができる。)
- CardDealer.cpp, 27 行目 の suit=CLUB や suit<=SPADE では、Suit 型から int 型への自動変換が行われる。
- CardDealer.cpp28 行目,32 行目, 33 行目 では、それぞれ int 型 →Suit 型, int 型 →Suit 型, int 型 →Rank 型 への変換を行いたい。これらの変換は自動では行なってくれないので、10∼12 行目 に配列 toSuit[], toRank[] を用意して変換の役割を負わせている。

(CardDealer クラスの利用) 配られた 5 枚のカードに関して、

◇ ストレートフラッシュ
⇔ (1 種類の suit, 5 種類の rank, 最大 rank∼(最大 rank−4) のカードが 1 枚ずつ),
または (1 種類の suit, 5 種類の rank, A と 2∼5 のカードが 1 枚ずつ)
◇ 4 カード ⇔ 4 枚揃った rank が 1 つある
◇ フルハウス ⇔ 上記以外, 2 種類の rank
◇ フラッシュ⇔ 上記以外, 1 種類の suit
◇ ストレート
⇔ (上記以外, 5 種類の rank, 最大 rank∼(最大 rank−4) のカードが 1 枚ずつ),
または (上記以外, 5 種類の rank, A と 2∼5 のカードが 1 枚ずつ)
◇ 3 カード ⇔ 上記以外, 3 枚揃った rank が 1 つある
◇ 2 ペア ⇔ 上記以外, 2 枚揃った rank が 2 つある
◇ 1 ペア ⇔ 上記以外, 2 枚揃った rank が 1 つある
◇ ノーペア ⇔ 上記以外

である。それゆえ、CardDealer クラスが上の様に定義されていれば、それを利用して例題 4.6 で要求されている作業を例えば次の C++プログラムの様に表すことができる。

```
[motoki@x205a]$ cat -n estimateProbOfDrawingPokerHand.cpp
     1 /* Poker のそれぞれの役が出る確率を MonteCarlo 手法で見積もる */
     2
     3 #include <iostream>
     4 #include <cstdlib>
     5 #include <ctime>
     6 #include "CardDealer.h"
     7 using namespace std;
     8
     9 int main()
    10 {
```

```
11  CardDealer dealer;
12  Card cardInHand[5];
13  int freqStraightFlush(0), freq4OfAKind(0), freqFullHouse(0),
14    freqFlush(0), freqStraight(0), freq3OfAKind(0),
15    freq2Pair(0), freq1Pair(0), freqNoPair(0);
16  int freqSuit[4], numOfSuitTypes;
17  int freqRank[15], numOfRankTypes, freqOfFreqRankValue[5], highestRank;
18
19  for (int i=1; i<=1000000000; ++i) {
20    dealer.dealCards(5, cardInHand);
21    //手配カードの suit や rank の分布を調べる
22    for (int k=CLUB; k<=SPADE; ++k)
23      freqSuit[k] = 0;
24    for (int k=RANK2; k<=ACE; ++k)
25      freqRank[k] = 0;
26    for (int k=0; k<5; ++k) {
27      ++freqSuit[cardInHand[k].suit];
28      ++freqRank[cardInHand[k].rank];
29    }
30    numOfSuitTypes = 0;
31    for (int k=CLUB; k<=SPADE; ++k) {
32      if (freqSuit[k] > 0)
33        ++numOfSuitTypes;
34    }
35    numOfRankTypes = 0;
36    highestRank = RANK2;
37    for (int k=0; k<5; ++k)
38      freqOfFreqRankValue[k] = 0;
39    for (int k=RANK2; k<=ACE; ++k) {
40      if (freqRank[k] > 0) {
41        ++numOfRankTypes;
42        highestRank = k;
43      }
44      ++freqOfFreqRankValue[freqRank[k]];
45    }
46    //手配カードの役を調べる
47    if (numOfSuitTypes==1 && numOfRankTypes==5
48        && freqRank[highestRank]==1 && freqRank[highestRank-1]==1
49        && freqRank[highestRank-2]==1 && freqRank[highestRank-3]==1
50        && freqRank[highestRank-4]==1)
51      ++freqStraightFlush;
52    else if (numOfSuitTypes==1 && numOfRankTypes==5
53             && freqRank[ACE]==1 && freqRank[2]==1 && freqRank[3]==1
54             && freqRank[4]==1 && freqRank[5]==1)
55      ++freqStraightFlush;
56    else if (freqOfFreqRankValue[4] == 1)
57      ++freq4OfAKind;
58    else if (numOfRankTypes == 2)
59      ++freqFullHouse;
60    else if (numOfSuitTypes==1)
61      ++freqFlush;
62    else if (numOfRankTypes==5
```

```
    63              && freqRank[highestRank]==1 && freqRank[highestRank-1]==1
    64              && freqRank[highestRank-2]==1 && freqRank[highestRank-3]==1
    65              && freqRank[highestRank-4]==1)
    66        ++freqStraight;
    67      else if (numOfRankTypes==5
    68              && freqRank[ACE]==1 && freqRank[2]==1 && freqRank[3]==1
    69              && freqRank[4]==1 && freqRank[5]==1)
    70        ++freqStraight;
    71      else if (freqOfFreqRankValue[3] == 1)
    72        ++freq3OfAKind;
    73      else if (freqOfFreqRankValue[2] == 2)
    74        ++freq2Pair;
    75      else if (freqOfFreqRankValue[2] == 1)
    76        ++freq1Pair;
    77      else
    78        ++freqNoPair;
    79      //時々、途中経過を画面に表示
    80      if (i%100000000 == 0) {
    81        time_t currentTime = time(NULL);
    82        cout << i << "-th hand's rank has been determined: "
    83             << ctime(&currentTime);
    84      }
    85    }
    86    //結果を表示
    87    cout << "ストレートフラッシュの確率の見積り："
    88         << freqStraightFlush/1e9 << endl
    89         << "4カード　の確率の見積り：" << freq4OfAKind/1e9 << endl
    90         << "フルハウスの確率の見積り：" << freqFullHouse/1e9 << endl
    91         << "フラッシュの確率の見積り：" << freqFlush/1e9 << endl
    92         << "ストレートの確率の見積り：" << freqStraight/1e9 << endl
    93         << "3カード　の確率の見積り：" << freq3OfAKind/1e9 << endl
    94         << "2ペア　　の確率の見積り：" << freq2Pair/1e9 << endl
    95         << "1ペア　　の確率の見積り：" << freq1Pair/1e9 << endl
    96         << "ノーペア　の確率の見積り：" << freqNoPair/1e9 << endl;
    97 }
[motoki@x205a]$ g++ estimateProbOfDrawingPokerHand.cpp CardDealer.cpp
[motoki@x205a]$ ./a.out
100000000-th hand's rank has been determined: Thu Oct 19 09:34:52 2017
200000000-th hand's rank has been determined: Thu Oct 19 09:35:14 2017
300000000-th hand's rank has been determined: Thu Oct 19 09:35:36 2017
400000000-th hand's rank has been determined: Thu Oct 19 09:35:58 2017
500000000-th hand's rank has been determined: Thu Oct 19 09:36:21 2017
600000000-th hand's rank has been determined: Thu Oct 19 09:36:43 2017
700000000-th hand's rank has been determined: Thu Oct 19 09:37:05 2017
800000000-th hand's rank has been determined: Thu Oct 19 09:37:27 2017
900000000-th hand's rank has been determined: Thu Oct 19 09:37:49 2017
1000000000-th hand's rank has been determined: Thu Oct 19 09:38:11 2017
ストレートフラッシュの確率の見積り：1.5332e-05
4カード　の確率の見積り：0.000239412
フルハウスの確率の見積り：0.00144001
フラッシュの確率の見積り：0.00196517
ストレートの確率の見積り：0.00392315
```

```
3カード　　の確率の見積り：0.0211279
2ペア　　の確率の見積り：0.0475396
1ペア　　の確率の見積り：0.422551
ノーペア　の確率の見積り：0.501198
[motoki@x205a]$
```

ここで、

- この利用例のプログラム13〜15行目では、手元のカード5枚がそれぞれの役になる回数をカウントする変数群を確保し、初期値0を設定することを宣言している。
- 16行目では、CLUBの枚数を保持する配列要素 (freqSuit[CLUB]), ..., suitの種類数を保持する変数 (NumOfSuitTypes) を宣言している。
- 17行目では、RANK2の枚数を保持する配列要素 (freqRank[RANK2]), ..., rankの種類数を保持する変数 (NumOfRankTypes), ..., 2枚揃ったrankの種類数を保持する配列要素 (freqFreqRankValue[2]), ..., 最大のrank値を保持する変数 (highestRank) を宣言している。
- 19行目に示されている様に、①ランダムに5枚のカードを配り②その役を調べる、という作業を10億回繰り返す。即実行終了という訳にも行かないので、実行終了を待つ不安を和らげるために、81〜85行目では1億回の繰り返しが終わる度にその由を標準出力に表示する様にしている。

演習問題

□演習 4.1 (N次元ベクトル空間の世界) 前ターム「プログラミングAI」例題11.3で作成したCプログラムをC++で実装し直すとどうなるか? どういうクラスを用意するかをよく考えた上で、(ほぼ) 同等のC++プログラムを作成せよ。

□演習 4.2 (2分木) 前ターム「プログラミングAI」例題12.4では、

① 入力ストリームに現れる 識別子 (1個以上の空白) 整数 という形のデータを読み込んでは、

② それを2分木上に追加登録する (但し、

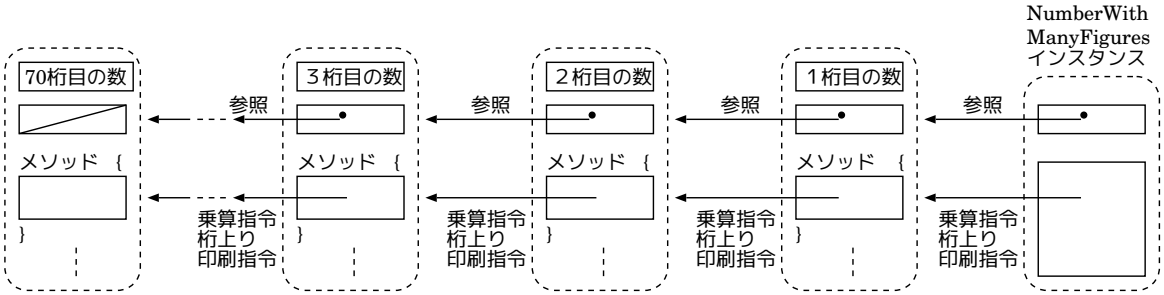
(左の子とその子孫の識別子) ≤ (親の識別子)

(親の識別子) < (右の子とその子孫の識別子)

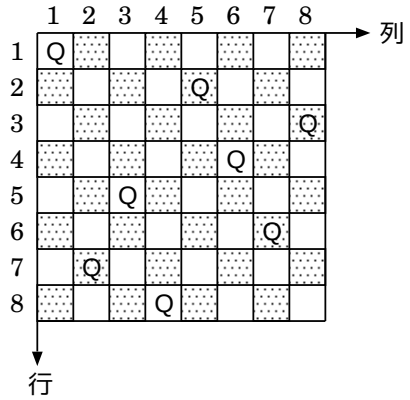
という関係を損なわない様に追加登録する)、

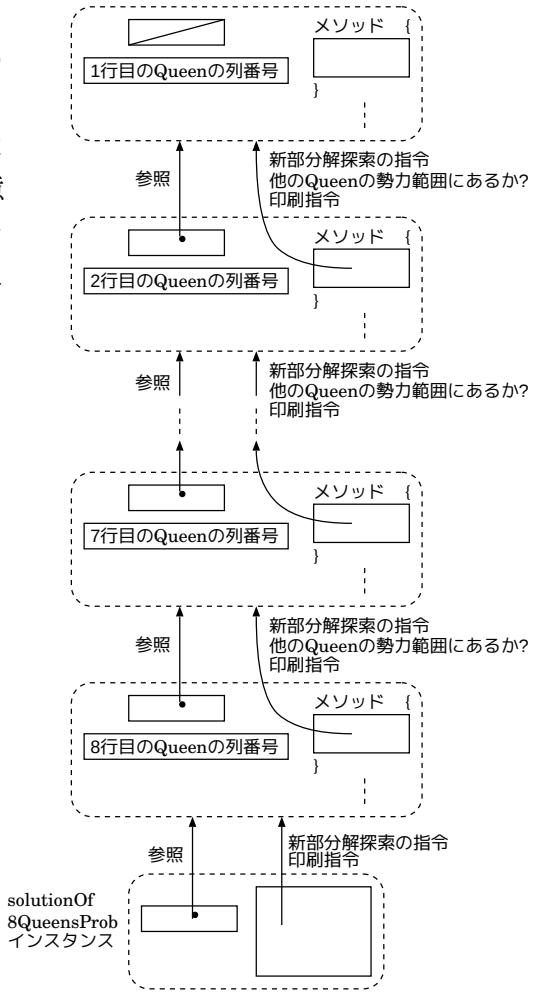
という作業を繰り返し、最後に2分木を中間順に走査し登録されたものを順に出力するCプログラムを作成した。これに対して、ここでは (ほぼ) 同等の処理を行うプログラムをC++で構成してみよ。

□演習 4.3 (53!の高精度計算) 53! を正確に計算して出力するC++プログラムを作成せよ。 但し、ここでは1桁分の数を記憶するオブジェクトを多数作り、それらの協調で計算を進めていくことにする。



□演習 4.4 (8-Queens 問題) チェス盤 (下図) の上に 8 個の Queen(i.e. 縦横斜めに攻撃能力があるチェスの駒) を互いに攻撃し合わない様に配置したい。各行に 1 つの Queen が配置されることになるので、各行に配置される Queen の列番号を記憶管理するオブジェクトを線形リスト状に繋げて、それらの間で、メッセージ交換をしてもらうことによって適切な配置を全て見つけ出すC++プログラムを作成せよ。





5　既定義クラスの拡張、多態性の実現、抽象クラス

- 既定義クラスを基にしたクラス定義
- 既定義クラスの拡張、is-a 関係
- 仮想関数を用いた多態性の実現
- 抽象クラス
- Makefile を用いた分割コンパイル, 例 (heapsort vs. bubblesort vs. llistsort, predator-prey simulation)

5.1　既定義クラスを基にしたクラス定義

{Pohl(1999)8.1 節,8.7.2 節, 柴田 (2014)4.1 節,4.3 節, プログラミング I(2017)22.5 節}

既定義クラスを利用したクラス定義、継承：　既に定義されたクラスを利用して新しいクラスを定義することができる。C++言語では、元になったクラスを**基底クラス** (base class) 、新しく定義したクラスを (元のクラスの) **派生クラス** (derived class) という。具体的には、C++言語では次の様に書く。

```
class [派生クラス名] : [アクセス指定子, 省略可] [基底クラス名] {
        新しいメンバの宣言、定義、など
}
```

もしくは、

```
struct [派生クラス名] : [アクセス指定子, 省略可] [基底クラス名] {
        新しいメンバの宣言、定義、など
}
```

ここで、

- これまで通り、キーワード `class` を用いた場合は新しく宣言されたメンバは暗黙に `private` として扱われ、キーワード `struct` を用いた場合は新しく宣言されたメンバは暗黙に `public` として扱われる。
- (既定義クラスを利用しない場合も含めて) 一般に、クラス定義で新しく宣言するメンバへのアクセス可否を指定するキーワードとして、`private`, `public` 以外に `protected` というものもある。これを指定した場合は、そのメンバは派生クラス (とその派生,...) のオブジェクトに対してのみ**限定公開**される。
- [基底クラス名]の前の [アクセス指定子, 省略可] の場所には `private`, `protected`, `public` のいずれかを指定できる。(省略時は `private` が指定されたものとして扱われる。)
  - ◇ `private` と指定された (または無指定の) 場合は、`private` **派生**と呼ばれる。この場合、基底クラス内のメンバの、派生クラスのメンバとしての扱いは次の様に設定される。

基底クラス内での指定		派生クラスのメンバとしての扱い	派生クラス内からのアクセス
`private`	⟶	`private`	不可
`protected`	⟶	`private`	可
`public`	⟶	`private`	可

従って、`private` 派生では、既存のクラスを内部で利用して全く新しいクラスを構成することになる。

◇ `protected` と指定された場合は、`protected` **派生**と呼ばれる。この場合、基底クラス内のメンバの、派生クラスのメンバとしての扱いは次の様に設定される。

基底クラス内での指定		派生クラスのメンバとしての扱い	派生クラス内からのアクセス
`private`	⟶	`private`	不可
`protected`	⟶	`protected`	可
`public`	⟶	`protected`	可

◇ `public` と指定された場合は、`public` **派生**と呼ばれる。この場合、基底クラス内のメンバの、派生クラスのメンバとしての扱いは次の様に設定される。

基底クラス内での指定		派生クラスのメンバとしての扱い	派生クラス内からのアクセス
`private`	⟶	`private`	不可
`protected`	⟶	`protected`	可
`public`	⟶	`public`	可

従って、`public` 派生では、既存のクラスを拡張して新しいクラスを構成することになる。基底クラスにおいて定められた (オブジェクトの) インタフェースは派生クラスにそのまま**継承** (inherit) される。

- 基底クラスで定義された コンストラクタ, デストラクタ, 関数 `operator=()` 以外の メンバは派生クラスに暗黙に**継承** (inherit) される。すなわち、基底クラスで定義された非 `static` なデータメンバについては、派生クラスのインスタンスにおいても領域確保され、また、基底クラスで定義された非 `static` なメンバ関数は、派生クラスの中で再定義／上書き (**オーバーライド**,override, という) されてなければ、派生クラスのメンバ関数として暗黙に使用することができる。

  ⟹ **継承をうまく利用すれば、同じ定義をあちこちのクラス定義の中で繰り返す手間はかなり省ける様になる。**

- 基底クラスで定義されたコンストラクタ, デストラクタは派生クラスに継承されない。

派生クラスのコンストラクタ： 派生クラスのコンストラクタは次の形に書く。

```
派生クラス名 ( 仮引数列 ) : 基底クラス名 ( 実引数列 ),
        新データメンバ名 ( 初期値の式 ), ..., 新データメンバ名 ( 初期値の式 ) {
    初期設定の仕上げ (必要な場合)
}
```

この中の、

- 「: 基底クラス名 (実引数列), ..., 新データメンバ名 (初期値の式) 」の部分は**初期化子リスト** (initializer list) と呼ばれるもので、データメンバの初期化をどう行うかを指示している。

- 「基底クラス名（実引数列）」の部分は**コンストラクタ初期化子** (constructor initializer) と呼ばれるもので、基底クラスのコンストラクタの呼び出しを指示している。(この部分を省略した場合はデフォルトコンストラクタの呼び出しとなる。)
- 実際の初期化は次の順に行われる。
  ① 基底クラスのコンストラクタの呼び出し
  ② データメンバの初期化 (初期化子リストに並んだ順ではなく、クラス定義内でメンバの並んだ順に初期化される。)
  ③ 派生クラスのコンストラクタの本体部

  この内、ステップ①,②により派生クラスのインスタンスが出来上がり、ステップ③によりその後の設定が為される。それゆえ、初期化子リストで省略を行うと①,②で支障が出る可能性がある。(e.g. コンストラクタ初期化子の記述を省略した場合、基底クラスのデフォルトコンストラクタが無いとコンパイルエラー。)

例 5.1 (色付き長方形のクラス,Rectangle クラスの拡張) 例題 4.2 (p.77) ではクラス定義の典型的な例として長方形を表すオブジェクトのクラス Rectangle を示した。ここでは、このクラスを拡張して色付き長方形を表すオブジェクトのクラス ColoredRectangle を定義してみよう。構成した派生クラスのコード ColoredRectangle.h, ColoredRectangle.cpp を次に示す。

```
[motoki@x205a]$ cat -n ColoredRectangle.h
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle（仕様部）*/
     2 /*（既定義の Rectangle クラス の拡張）                          */
     3
     4 #ifndef ___Class_ColoredRectangle
     5 #define ___Class_ColoredRectangle
     6
     7 #include <string>
     8 #include "Rectangle.h"
     9
    10 class ColoredRectangle : public Rectangle {
    11   std::string color;
    12 public:
    13   ColoredRectangle(double width = 0.0, double height = 0.0,
    14                    std::string color = "black")
    15     : Rectangle(width, height), color(color) {}
    16   ColoredRectangle(const ColoredRectangle& rectangle);
                                                        //コピーコンストラクタ
    17   // オブジェクト (もしくはクラス全体) の情報を提供するための関数群 (追加)
    18   std::string getColor() const { return color; }
    19   void setColor(std::string color) { this->color = color; }
    20   std::string toString() const; //内部の長方形情報を string データとして返す
    21 };
    22
    23 #endif
[motoki@x205a]$ cat -n ColoredRectangle.cpp
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle（実装部）*/
     2 /*（既定義の Rectangle クラス の拡張）                          */
     3
     4 #include <sstream>
```

```
 5 #include <string>
 6 #include "ColoredRectangle.h"
 7 using namespace std;
 8
 9 // 各種コンストラクタ -----------------------------------------
10 ColoredRectangle::ColoredRectangle(const ColoredRectangle& rectangle)
                                                    //コピーコンストラクタ
11                   : Rectangle(rectangle.getWidth(), rectangle.getHeight()),
12                     color(rectangle.color) {}
13
14 // ColoredRectangle型オブジェクトを操作するための関数群 ----------------------
15 // オブジェクト内部に保持している長方形情報を string 型データとして返す
16 string ColoredRectangle::toString() const
17 {
18   ostringstream ostr;
19   ostr << "rectangle(id=" << getId()
20        << ",width=" << getWidth() << ",height=" << getHeight()
21        << ",color=\"" << color << "\")";
22   return ostr.str();
23 }
[motoki@x205a]$
```

これに関して、

- 基底クラスを「拡張」するので、上の ColoredRectangle.h,10 行目 で明示している様に基底クラスを public **派生**させる。
- 上の ColoredRectangle.h,15 行目 の「: Rectangle(width, height), color(color)」の部分、ColoredRectangle.cpp,11∼12 行目 の「: Rectangle(rectangle.getWidth(), rectangle.getHeight()), color(rectangle.color)」の部分が初期化子のリストで、それらの中の「Rectangle(width, height)」、「Rectangle(rectangle.getWidth(), rectangle.getHeight())」がそれぞれ基底クラスのコンストラクタを呼び出している部分である。
- 上の ColoredRectangle.h,20 行目 と ColoredRectangle.cpp,16∼23 行目 で定義されているメンバ関数 toString() と同じシグニチャの関数が、基底クラスでも定義されている。この場合、基底クラスでの定義は派生クラスにおいては新しいものに置き換えられる (i.e. **オーバーライド**される)。
- 上のクラス ColoredRectangle が定義されていれば、それを使って次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n useColoredRectangle.cpp
 1 /* ColoredRectangle.h, ColoredRectangle.cpp の利用例 */
 2
 3 #include <iostream>
 4 #include <string>
 5 #include "ColoredRectangle.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10   ColoredRectangle rect;
11
```

```
12   cout << "(1)" << rect.toString() << endl
13        << "  |rect.getId()=" << rect.getId()
14        <<       ", rect.getWidth()=" << rect.getWidth() << endl
15        << "  |rect.getHeight()=" << rect.getHeight()
16        <<       ", rect.getColor()=\"" << rect.getColor() << "\"" << endl
17        << "  |rect.getArea()=" << rect.getArea() << endl;
18   rect.setWidth(2.0);
19   rect.setHeight(3.0);
20   rect.setColor("blue");
21   cout << "(2)" << rect.toString() << endl
22        << "  |rect.getId()=" << rect.getId()
23        <<       ", rect.getWidth()=" << rect.getWidth() << endl
24        << "  |rect.getHeight()=" << rect.getHeight()
25        <<       ", rect.getColor()=\"" << rect.getColor() << "\"" << endl
26        << "  |rect.getArea()=" << rect.getArea() << endl;
27
28   //cout << "(3)" << (new ColoredRectangle(4.0, 5.0, "red"))->toString()
     //     << endl;           // heap 領域のメモリが開放されないままになる
29   ColoredRectangle* ptrRect = new ColoredRectangle(4.0, 5.0, "red");
30   cout << "(3)" << ptrRect->toString() << endl;
31   delete ptrRect;
32 }
[motoki@x205a]$ g++ useColoredRectangle.cpp ColoredRectangle.cpp Rectangle.cpp
[motoki@x205a]$ ./a.out
(1)rectangle(id=0,width=0,height=0,color="black")
  |rect.getId()=0, rect.getWidth()=0
  |rect.getHeight()=0, rect.getColor()="black"
  |rect.getArea()=0
(2)rectangle(id=0,width=2,height=3,color="blue")
  |rect.getId()=0, rect.getWidth()=2
  |rect.getHeight()=3, rect.getColor()="blue"
  |rect.getArea()=6
(3)rectangle(id=1,width=4,height=5,color="red")
[motoki@x205a]$
```

ここで、

◇ この利用例のプログラム 13~15 行目,17~19 行目, 22~24 行目,26 行目 では、Colored Rectangle.h 内に記述されてないにもかかわらず getId(), getWidth(), getHeight(), getArea(), setWidth(), setHeight() がメンバ関数として使われている。これは、基底クラスである Rectangle のメンバ関数を継承できているからである。

例 5.2 (色付き長方形のクラス,protected 指定子を用いて Rectangle クラスを定義した場合)
例題 4.2 (p.77) で与えた Rectangle クラスのコードでは、将来の派生を全く考えずにデータメンバを全て非公開 (private) とした。しかし、将来の派生を考慮に入れてデータメンバを限定公開 (protected) する選択肢もある。そのコードを次に示す。

```
[motoki@x205a]$ cat -n Rectangle_verProt.h
 1 /* 長方形オブジェクトのクラス Rectangle (仕様部) */
 2 /* (protected データ版)                          */
 3
 4 #ifndef ___Class_Rectangle
```

```
 5 #define ___Class_Rectangle
 6
 7 #include <string>
 8
 9 class Rectangle {
10 protected:
11   static int numOfInstances;  // これまでに生成したインスタンスの個数
12   const int id;     // 長方形インスタンスに付ける id 番号
13   double width;     // 長方形の横幅
14   double height;    // 長方形の高さ
15 public:
16   Rectangle(double width = 0.0, double height = 0.0)
17     : id(numOfInstances++), width(width), height(height) {}
18   Rectangle(const Rectangle& rectangle);     //コピーコンストラクタ
19   //~Rectangle() {}
20   // オブジェクト (もしくは Rectangle クラス全体) の情報を提供するための関数群
21   static int getNumOfInstances() { return numOfInstances; }
22                       //これまでに生成した Rectangle インスタンスの個数を返す
23   double getId() const { return id; }
24   double getWidth() const { return width; }
25   double getHeight() const { return height; }
26   void setWidth(double width) { this->width = width; }
27   void setHeight(double height) { this->height = height; }
28   double getArea() const { return width*height; }  //長方形の面積を返す
29   std::string toString() const; //内部の長方形情報を string データとして返す
30 };
31
32 #endif
[motoki@x205a]$ cat -n Rectangle_verProt.cpp
 1 /* 長方形オブジェクトのクラス Rectangle (実装部) */
 2 /* (protected データ版)                          */
 3
 4 #include <sstream>
 5 #include <string>
 6 #include "Rectangle_verProt.h"
 7 using namespace std;
 8
 9 // static 変数の初期化 ------------------------------------------
10 int Rectangle::numOfInstances = 0;
11
12 // 各種コンストラクタ -------------------------------------------
13 Rectangle::Rectangle(const Rectangle& rectangle)   // コピーコンストラクタ
14           : id(numOfInstances++),
15     width(rectangle.width), height(rectangle.height) {}
16
17 // Rectangle 型オブジェクトを操作するための関数群 ----------------------
18 // オブジェクト内部に保持している長方形情報を string 型データとして返す
19 string Rectangle::toString() const
20 {
21   ostringstream os;
22   os << "rectangle(id=" << id
23      << ",width=" << width << ",height=" << height << ")";
```

```
    24   return os.str();
    25 }
[motoki@x205a]$
```

例題4.2で示したコードとの本質的な違いは、Rectangle_verProt.h,10行目のprotected:という指定だけである。これらの Rectangle_verProt.h, Rectangle_verProt.cpp を利用して色付き長方形を表すオブジェクトのクラス ColoredRectangle を派生させたコードとしては、次の様なものを考えることができる。

```
[motoki@x205a]$ cat -n ColoredRectangle_verProtBase.h
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle (仕様部) */
     2 /* (既定義の Rectangle クラス の拡張)                          */
     3
     4 #ifndef ___Class_ColoredRectangle
     5 #define ___Class_ColoredRectangle
     6
     7 #include <string>
     8 #include "Rectangle_verProt.h"
     9
    10 class ColoredRectangle : public Rectangle {
    11 protected:
    12   std::string color;
    13 public:
    14   ColoredRectangle(double width = 0.0, double height = 0.0,
    15                    std::string color = "black")
    16     : Rectangle(width, height), color(color) {}
    17   ColoredRectangle(const ColoredRectangle& rectangle);
                                                        //コピーコンストラクタ
    18   // オブジェクト (もしくはクラス全体) の情報を提供するための関数群 (追加)
    19   std::string getColor() const { return color; }
    20   void setColor(std::string color) { this->color = color; }
    21   std::string toString() const; //内部の長方形情報を string データとして返す
    22 };
    23
    24 #endif
[motoki@x205a]$ cat -n ColoredRectangle_verProtBase.cpp
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle (実装部) */
     2
     3 #include <sstream>
     4 #include <string>
     5 #include "ColoredRectangle_verProtBase.h"
     6 using namespace std;
     7
     8 // 各種コンストラクタ ------------------------------------------
     9 ColoredRectangle::ColoredRectangle(const ColoredRectangle& rectangle)
                                                        //コピーコンストラクタ
    10                   : Rectangle(rectangle.width, rectangle.height),
    11                     color(rectangle.color) {}
    12
    13 // ColoredRectangle 型オブジェクトを操作するための関数群 ----------------------
    14 // オブジェクト内部に保持している長方形情報を string 型データとして返す
    15 string ColoredRectangle::toString() const
    16 {
```

```
   17   ostringstream ostr;
   18   ostr << "rectangle(id=" << id
   19        << ",width=" << width << ",height=" << height
   20        << ",color=\"" << color << "\")";
   21   return ostr.str();
   22 }
[motoki@x205a]$
```

これに関して、

- 例題4.2で与えたコードColoredRectangle_verProtBase.h, ColoredRectangle_verProtBase.cpp との本質的な違いは、実装部のColoredRectangle_verProtBase.cpp,10 行目, 18~19 行目 でメンバ関数getWidth(), getHeight(), getId() を呼ばずにそれぞれ直接 width, height, id を参照している箇所だけである。protected 指定によりこういった直接の参照が可能になっている。
- クラス ColoredRectangle がここでの例の様に定義されていた場合も、クラス自体に備わったメンバの仕様は例 5.1 のものと同じなので、当然、例 5.1 と同様の使用が可能である。すなわち、

```
[motoki@x205a]$ cat -n useColoredRectangle_verProtBase.cpp
     1 /* ColoredRectangle_verProtBase.h,           */
     2 /* ColoredRectangle_verProtBase.cpp の利用例 */
     3
     4 #include <iostream>
     5 #include <string>
     6 #include "ColoredRectangle_verProtBase.h"
     7 using namespace std;
     8
     9 int main()
    10 {
    11   ColoredRectangle rect;
    12
    13   cout << "(1)" << rect.toString() << endl
    14        << "  |rect.getId()=" << rect.getId()
    15        <<       ", rect.getWidth()=" << rect.getWidth() << endl
    16        << "  |rect.getHeight()=" << rect.getHeight()
    17        <<       ", rect.getColor()=\"" << rect.getColor() << "\"" << endl
    18        << "  |rect.getArea()=" << rect.getArea() << endl;
    19   rect.setWidth(2.0);
    20   rect.setHeight(3.0);
    21   rect.setColor("blue");
    22   cout << "(2)" << rect.toString() << endl
    23        << "  |rect.getId()=" << rect.getId()
    24        <<       ", rect.getWidth()=" << rect.getWidth() << endl
    25        << "  |rect.getHeight()=" << rect.getHeight()
    26        <<       ", rect.getColor()=\"" << rect.getColor() << "\"" << endl
    27        << "  |rect.getArea()=" << rect.getArea() << endl;
    28
    29   //cout << "(3)" << (new ColoredRectangle(4.0, 5.0, "red"))->toString()
         //     << endl;           // heap 領域のメモリが開放されないままになる
    30   ColoredRectangle* ptrRect = new ColoredRectangle(4.0, 5.0, "red");
    31   cout << "(3)" << ptrRect->toString() << endl;
    32   delete ptrRect;
```

```
    33 }
[motoki@x205a]$ g++ useColoredRectangle_verProtBase.cpp
                    ColoredRectangle_verProtBase.cpp Rectangle_verProt.cpp
[motoki@x205a]$ ./a.out
(1)rectangle(id=0,width=0,height=0,color="black")
  |rect.getId()=0, rect.getWidth()=0
  |rect.getHeight()=0, rect.getColor()="black"
  |rect.getArea()=0
(2)rectangle(id=0,width=2,height=3,color="blue")
  |rect.getId()=0, rect.getWidth()=2
  |rect.getHeight()=3, rect.getColor()="blue"
  |rect.getArea()=6
(3)rectangle(id=1,width=4,height=5,color="red")
[motoki@x205a]$
```

5.2 既定義クラスの拡張、is-a 関係
—public 派生の場合の 基底クラス–派生クラス間の関係—

{Pohl(1999)8.2 節, 柴田 (2014)4.2 節 }

public 派生の場合のインタフェース継承： public 派生の場合、基底クラスのオブジェクトに備わっていたインタフェースは全て派生クラスのオブジェクトにもそのまま引き継がれるので、派生によってインタフェースの拡張されたオブジェクトのクラスが定義されることになる。

データ型としてのクラス： クラス定義は然るべきインタフェースを備えたオブジェクトを作り出すための型枠であるが、クラスをデータ型と見る場合は、そのデータ型に属するために必要なインタフェースを規定したものであると考えることができる。すなわち、

$$\text{定義されたクラスの型に属するオブジェクトの集合} \stackrel{\text{def}}{=} \left(\begin{array}{l}\text{そのクラスのインスタンスに備わったインタフェース}\\ \qquad\text{と同等のインタフェースを有するオブジェクトの集合}\end{array}\right)$$

と考える。ここで、特に public 派生の場合は、基底クラスのオブジェクトに備わっていたインタフェースは全て派生クラスのオブジェクトにもそのまま引き継がれるので、

$$\text{派生クラスの型に属するオブジェクトの集合} \subseteq \text{基底クラスの型に属するオブジェクトの集合、}$$

すなわち、**派生クラス型のオブジェクトは基底クラス型オブジェクトの一種**ということになる。更に、この包含関係を念頭に置いて「派生クラス型は基底クラス型の**サブタイプ** (subtype, 亜種) である」と言うこともある。(**補足：** この言い方に従った場合、「int 型で表せるデータの集合 ⊆double 型で表せるデータの集合」であるので、int 型は double 型のサブタイプということになる。)

is-a 関係, has-a 関係： 一般に、オブジェクト指向ではクラス間の次の 2 つの関係に注目する。

- **is-a 関係** ··· クラス A のインスタンスがクラス B のインスタンスの一種になる時、クラス A,B の間に is-a 関係があるという。こういう場合には、C++言語では、

クラス **B** を public 派生してクラス **A** を定義するのが良い。

- **has-a 関係** ··· クラス A のインスタンスがクラス B のインスタンスを一部分として持つ時、クラス A,B の間に has-a 関係があるという。こういう場合には、クラス **A** のデータメンバとしてクラス **B** のインスタンスを保持 (または参照) する領域を用意するのが妥当である。

クラス型間のサブタイプの関係を C++プログラム上にどう反映させるか： public 派生の場合の「派生クラス型オブジェクトの集合 ⊆ 基底クラス型オブジェクトの集合」という関係をプログラム上に反映させるために、C++言語では、

> 基底クラス型へのポインタ型変数に
> public 派生クラス型オブジェクトへのポインタを保持できる

様になっている。(但し、その際、変数への代入の前に基底クラス型へのポインタ型に**暗黙に型変換**される。)

例 5.3 (ColoredRectangle*型ポインタ値を Rectangle*型変数に保持) 例題 4.2 (p.77), 例 5.1 (p.105) で考えた基底クラス Rectangle と (public) 派生クラス ColoredRectangle を用いて

> 基底クラス型へのポインタ型変数に
> public 派生クラス型オブジェクトへのポインタを保持できる、

ことを確認した例を次に示す。

```
[motoki@x205a]$ cat -n testStoringPtrToDerivObjInBaseVar.cpp
     1 /* 基底クラス型変数に派生クラスオブジェクトへのポインタを保持  */
     2 /* した時の動作確認                                          */
     3
     4 #include <iostream>
     5 #include "ColoredRectangle.h"
     6 using namespace std;
     7
     8 int main()
     9 {
    10   ColoredRectangle* ptrToDerivedClassObj = new ColoredRectangle(1.0, 2.0);
    11   cout << ptrToDerivedClassObj->toString() <<endl;
    12
    13   Rectangle* ptrToBaseClassObj(ptrToDerivedClassObj);
    14   cout << ptrToBaseClassObj->toString() <<endl;
    15 }
[motoki@x205a]$ g++ testStoringPtrToDerivObjInBaseVar.cpp
                                          ColoredRectangle.cpp Rectangle.cpp
[motoki@x205a]$ ./a.out
rectangle(id=0,width=1,height=2,color="black")
rectangle(id=0,width=1,height=2)
[motoki@x205a]$
```

これに関して、

- プログラム 13 行目 は、変数 ptrToBaseClassObj への初期設定を行なっている行で、

  ```
  Rectangle* ptrToBaseClassObj = ptrToDerivedClassObj;
  ```

  と書くのと同じである。

- プログラム 14 行目 の出力結果を見て分かる様に、変数 ptrToBaseClassObj が保持しているポインタの先にあるのは派生クラス ColoredRectangle 型オブジェクトであるが、型変換されて保持されているため、ptrToBaseClassObj->toString() によって基底クラス Rectangle 内で定められたメンバ関数 toString() が呼び出されている。

例 5.4 (private 派生の場合) ここでは、private 派生の場合に先の例 5.3 で確認した

> 基底クラス型へのポインタ型変数に
> public 派生クラス型オブジェクトへのポインタを保持できる、

ということがどうなるかを調べたい。そのために、例題 4.2 (p.77) で示した Rectangle クラスを private 派生させ、先の例 5.3 と同様のことを行なってみた結果を次に示す。

```
[motoki@x205a]$ cat -n ColoredRectangle_verPrivDeriv.h
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle (仕様部) */
     2 /* (既定義の Rectangle クラス を private 派生した版)             */
     3
     4 #ifndef ___Class_ColoredRectangle
     5 #define ___Class_ColoredRectangle
     6
     7 #include <string>
     8 #include "Rectangle.h"
     9
    10 class ColoredRectangle : private Rectangle {
    11   std::string color;
    12 public:
    13   ColoredRectangle(double width = 0.0, double height = 0.0,
    14                    std::string color = "black")
    15     : Rectangle(width, height), color(color) {}
    16   ColoredRectangle(const ColoredRectangle& rectangle);
                                                        //コピーコンストラクタ
    17   // オブジェクト (もしくはクラス全体) の情報を提供するための関数群 (追加)
    18   std::string getColor() const { return color; }
    19   void setColor(std::string color) { this->color = color; }
    20   std::string toString() const; //内部の長方形情報を string データとして返す
    21 };
    22
    23 #endif
[motoki@x205a]$ cat -n ColoredRectangle_verPrivDeriv.cpp
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle (実装部) */
     2 /* (既定義の Rectangle クラス を private 派生した版)             */
     3
     4 #include <sstream>
     5 #include <string>
     6 #include "ColoredRectangle_verPrivDeriv.h"
     7 using namespace std;
     8
     9 // 各種コンストラクタ ------------------------------------------
    10 ColoredRectangle::ColoredRectangle(const ColoredRectangle& rectangle)
                                                        //コピーコンストラクタ
    11                  : Rectangle(rectangle.getWidth(), rectangle.getHeight()),
    12                    color(rectangle.color) {}
    13
```

```
    14 // ColoredRectangle型オブジェクトを操作するための関数群 ----------------------
    15 // オブジェクト内部に保持している長方形情報を string 型データとして返す
    16 string ColoredRectangle::toString() const
    17 {
    18   ostringstream ostr;
    19   ostr << "rectangle(id=" << getId()
    20        << ",width=" << getWidth() << ",height=" << getHeight()
    21        << ",color=\"" << color << "\")";
    22   return ostr.str();
    23 }
[motoki@x205a]$ cat -n testStoringPtrToDerivObjInBaseVar_whenPrivDeriv.cpp
     1 /* private 派生した場合、                                              */
     2 /* 基底クラス型変数に派生クラスオブジェクトへのポインタを保持 */
     3 /* した時の動作確認                                                   */
     4
     5 #include <iostream>
     6 #include "ColoredRectangle_verPrivDeriv.h"
     7 using namespace std;
     8
     9 int main()
    10 {
    11   ColoredRectangle* ptrToDerivedClassObj = new ColoredRectangle(1.0, 2.0);
    12   cout << ptrToDerivedClassObj->toString() <<endl;
    13
    14   Rectangle* ptrToBaseClassObj;      // <---この宣言自体はエラーにならない
    15   ptrToBaseClassObj = ptrToDerivedClassObj;
    16 }
[motoki@x205a]$ g++ testStoringPtrToDerivObjInBaseVar_whenPrivDeriv.cpp
                                ColoredRectangle_verPrivDeriv.cpp Rectangle.cpp
testStoringPtrToDerivObjInBaseVar_whenPrivDeriv.cpp: 関数 ‘int main()’ 内:
testStoringPtrToDerivObjInBaseVar_whenPrivDeriv.cpp:15:21: エラー:‘Rectangle’is an
 inaccessible base of ‘ColoredRectangle’
   ptrToBaseClassObj = ptrToDerivedClassObj;
                     ^
[motoki@x205a]$
```

これに関して、

- 例5.1, 例5.3 のコードとの本質的な違いは、上の ColoredRectangle_verPrivDeriv.h, 11 行目 の private 指定と、testStoringPtrToDerivObjInBaseVar_whenPrivDeriv.cpp, 14∼15 行目 の「Rectangle* ptrToBaseClassObj;」という変数宣言,「ptrToBaseClassObj = ptrToDerivedClassObj;」という代入文だけである。
- コンパイル結果を見ると、testStoringPtrToDerivObjInBaseVar_whenPrivDeriv.cpp, 15 行目 の代入文において、「‘Rectangle’ is an inaccessible base of ‘ColoredRectangle’」という理由で (ポインタの型変換がうまくいかずに) コンパイルエラーとなっていることが分かる。(**補足：** private 派生なので、生成した ColoredRectangle 型オブジェクトは Rectangle 型オブジェクトとして備えているはずのインタフェースを保有していない。そのため、ColoredRectangle 型オブジェクトを Rectangle 型と見做して使うことは出来ない。)

5.3 仮想関数を用いた多態性の実現

{Pohl(1999)8.3 節, 柴田 (2014)5.1 節 }

多態性： 一般に、1 つの変数や関数 (または関数呼出し) が実行時の状況により色々に振舞える能力を**多態性** (**多相性**, **ポリモルフィズム**,polymorphism) と呼んでいる。一般のプログラミング言語で、実行の状況に応じて色々なデータ型の値を持てる変数や関数引数等を**多態変数**, **多態引数**などと呼び、多態引数を持つ関数を **(純) 多態関数**と呼ぶ。(C 言語ではどの変数もデータ型が固定されているのでこういうことは起こらないが、Lisp の様に変数に動的にデータ型を割り付ける言語ではあり得る。) 純多態関数の実体 (コード) は 1 つで、その中で実引数のデータ型に応じて適切な処理内容が選択されることになる。一方、同様の選択・自動切り替えを実現するために、1 つの関数名や演算記号に対して引数のデータ型毎に別々の関数本体, 演算処理が定義されることもある。これも多態性の一種で、**多重定義** (overloading) または**場当たり的多態性**と呼ぶ。これらの多態性によって、多態関数を呼び出す側では多態引数のデータ型に応じて呼び出す関数を切り替える必要はなくなる。

例 5.5 (C 言語における演算記号の多重定義) C 言語において、整数の加算を行う時も実数の加算を行う時も、(実際の処理の中身は異るが) + という共通の記号を用いる。これは非オブジェクト指向言語における代表的な多重定義・多態性の例になっている。

C++言語における多態性の実現：

- **関数の多重定義** ··· C++言語において関数の多重定義ができることは、既に 2.11 節で説明された通りである。⟶ 例 2.3
- **演算子の多重定義** ··· ビット列の左シフトを表す演算子 << は、演算子の左側に ostream オブジェクトが来た時はストリーム出力を行う様に多重定義されている。また、通常は加算演算子として用いている + は、演算子の左側に string オブジェクトが来た時は文字列を繋げる働きをする様に多重定義されている。この様な演算子の多重定義は、例 4.3 や例 4.4 に見られる様に、関数の多重定義の一種としてそれぞれのユーザが独自に導入することもできる。
- **多態変数** ··· 前節 5.2 で説明された
  > 基底クラス型へのポインタ型変数に
  > public 派生クラス型オブジェクトへのポインタを保持できる、

  ということから判断して、基底クラス型へのポインタ型変数は多態変数と見做せないこともない。ただ、型変換されて保持するので、厳密な意味では多態変数とは言えない。(C++言語でも、それぞれの変数のデータ型は固定されているので、多態変数はあり得ない。)
- **関数呼び出しの多態性** ··· やはり、前節 5.2 で説明された
  > 基底クラス型へのポインタ型変数に
  > public 派生クラス型オブジェクトへのポインタを保持できる、

  ということに注目する。すなわち、

  ```
  BaseClass* ptrToBaseClassObj;
  ```

  と宣言されていれば、この変数 ptrToBaseClassObj は BaseClass を public 派生して得られたクラス、あるいはそれを更に public 派生して得られたクラス、... に属するオブジェクトへのポインタを保持できる、ということに注目する。確か

に保持する際に型変換が行われるが、ポインタの指している先には色々なサブタイプ (派生クラス) のオブジェクトが実体として存在している。**C++言語では、このことを利用して、関数呼び出しの多態性を実現することができる。**具体的には、基底クラスにおいてメンバ関数を定義する際に virtual というキーワードが指定され、例えば

virtual データ型 funcName (仮引数列);

と宣言されていた場合、

ptrToBaseClassObj-> funcName (実引数列)

というコードを書くと、この部分は「BaseClass に備わったメンバ関数funcName() の固定的な呼び出し」ではなく、

> ポインタ ptrToBaseClassObj の指している**派生クラス**
> **に備わった**メンバ関数 funcName() の動的な呼び出し

という形で多態的に処理される様になる。すなわち、プログラム実行時に動的にポインタ ptrToBaseClassObj の指している (型変換前の) 本来のクラスのメンバ関数 funcName() を特定した上で、それを実行してくれる様になる。virtual 指定された (メンバ) 関数を**仮想関数**と呼ぶ。基底クラスにおける virtual 指定は派生クラスに暗黙に継承されるので、派生クラスにおいての重ねての指定は不要である。

例 5.6 (仮想関数, 多態的なメンバ関数呼び出し) キーワード virtual の効果、すなわち

> 基底クラスBaseClass でメンバ関数に virtual 指定すると、
> BaseClass*型変数を通じたそのメンバ関数の呼び出しは、
> そのポインタの指すオブジェクトの本当のデータ型を認識した上で、
> そのデータ型 (派生クラス) に合ったメンバ関数の動的な選択・実行をもたらす、

ということを確認するために、例 5.3 で考えた処理を、基底クラス Rectangle 内のメンバ関数 toString() に virtual 指定を付けた上で、再度実行してみよう。まず、クラス Rectangle, ColoredRectangle の定義は次の通り。

```
[motoki@x205a]$ cat -n Rectangle_verVirtualToString.h
     1 /* 長方形オブジェクトのクラス Rectangle (仕様部)   */
     2 /* (toString() を virtual 化して多態的に振舞わせる版) */
     3
     4 #ifndef ___Class_Rectangle
     5 #define ___Class_Rectangle
     6
     7 #include <string>
     8
     9 class Rectangle {
    10   static int numOfInstances;  // これまでに生成したインスタンスの個数
    11   const int id;     // 長方形インスタンスに付ける id 番号
    12   double width;     // 長方形の横幅
    13   double height;    // 長方形の高さ
    14 public:
    15   Rectangle(double width = 0.0, double height = 0.0)
    16     : id(numOfInstances++), width(width), height(height) {}
    17   Rectangle(const Rectangle& rectangle);     //コピーコンストラクタ
    18   //~Rectangle() {}
    19   // オブジェクト (もしくは Rectangle クラス全体) の情報を提供するための関数群
```

```
    20   static int getNumOfInstances() { return numOfInstances; }
    21                       //これまでに生成した Rectangle インスタンスの個数を返す
    22   double getId() const { return id; }
    23   double getWidth() const { return width; }
    24   double getHeight() const { return height; }
    25   void setWidth(double width) { this->width = width; }
    26   void setHeight(double height) { this->height = height; }
    27   double getArea() const { return width*height; }  //長方形の面積を返す
    28   virtual std::string toString() const;
                                //内部保持の長方形情報を string データとして返す
    29 };
    30
    31 #endif
[motoki@x205a]$ cat -n Rectangle_verVirtualToString.cpp
     1 /* 長方形オブジェクトのクラス Rectangle（実装部）   */
     2 /* (toString() を virtual 化して多態的に振舞わせる版) */
     3
     4 #include <sstream>
     5 #include <string>
     6 #include "Rectangle_verVirtualToString.h"
     7 using namespace std;
     8
     9 // static 変数の初期化 ------------------------------------------
    10 int Rectangle::numOfInstances = 0;
    11
    12 // 各種コンストラクタ ------------------------------------------
    13 Rectangle::Rectangle(const Rectangle& rectangle)   // コピーコンストラクタ
    14           : id(numOfInstances++),
    15             width(rectangle.width), height(rectangle.height) {}
    16
    17 // Rectangle 型オブジェクトを操作するための関数群 ----------------------
    18 // オブジェクト内部に保持している長方形情報を string 型データとして返す
    19 string Rectangle::toString() const
    20 {
    21   ostringstream os;
    22   os << "rectangle(id=" << id
    23      << ",width=" << width << ",height=" << height << ")";
    24   return os.str();
    25 }
[motoki@x205a]$ cat -n ColoredRectangle_verVirtualToString.h
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle（仕様部）*/
     2 /* (既定義 (toString() を virtual 化) の Rectangle クラス の拡張    */
     3
     4 #ifndef ___Class_ColoredRectangle
     5 #define ___Class_ColoredRectangle
     6
     7 #include <string>
     8 #include "Rectangle_verVirtualToString.h"
     9
    10 class ColoredRectangle : public Rectangle {
    11   std::string color;
    12 public:
```

```
    13   ColoredRectangle(double width = 0.0, double height = 0.0,
    14                    std::string color = "black")
    15     : Rectangle(width, height), color(color) {}
    16   ColoredRectangle(const ColoredRectangle& rectangle);
                                                        //コピーコンストラクタ
    17   // オブジェクト (もしくはクラス全体) の情報を提供するための関数群 (追加)
    18   std::string getColor() const { return color; }
    19   void setColor(std::string color) { this->color = color; }
    20   std::string toString() const; //内部の長方形情報を string データとして返す
    21 };
    22
    23 #endif
[motoki@x205a]$ cat -n ColoredRectangle_verVirtualToString.cpp
     1 /* 色付き長方形オブジェクトのクラス ColoredRectangle (実装部) */
     2 /* (既定義 (toString() を virtual 化) の Rectangle クラス の拡張     */
     3
     4 #include <sstream>
     5 #include <string>
     6 #include "ColoredRectangle_verVirtualToString.h"
     7 using namespace std;
     8
     9 // 各種コンストラクタ -------------------------------------------
    10 ColoredRectangle::ColoredRectangle(const ColoredRectangle& rectangle)
                                                          //コピーコンストラクタ
    11                   : Rectangle(rectangle.getWidth(), rectangle.getHeight()),
    12                     color(rectangle.color) {}
    13
    14 // ColoredRectangle型オブジェクトを操作するための関数群 ----------------------
    15 // オブジェクト内部に保持している長方形情報を string 型データとして返す
    16 string ColoredRectangle::toString() const
    17 {
    18   ostringstream ostr;
    19   ostr << "rectangle(id=" << getId()
    20        << ",width=" << getWidth() << ",height=" << getHeight()
    21        << ",color=\"" << color << "\")";
    22   return ostr.str();
    23 }
[motoki@x205a]$
```

これに関して、

- 例 5.3 との本質的な違いは、Rectangle_verVirtualToString.h,28 行目 の virtual 指定だけである。
- 上の様にクラス Rectangle, CploredRectangle が定義されていた場合は、例 5.3 で考えた main() の処理は次の様に進む。

```
[motoki@x205a]$ cat -n testStoringPtrToDerivObjInVirtualBaseVar.cpp
     1 /* toString() を virtual 化した上で、                              */
     2 /* 基底クラス型変数に派生クラスオブジェクトへのポインタを保持  */
     3 /* した時の動作確認                                              */
     4
     5 #include <iostream>
     6 #include "ColoredRectangle_verVirtualToString.h"
     7 using namespace std;
```

```
 8
 9 int main()
10 {
11   ColoredRectangle* ptrToDerivedClassObj = new ColoredRectangle(1.0, 2.0);
12   cout << ptrToDerivedClassObj->toString() <<endl;
13
14   Rectangle* ptrToBaseClassObj(ptrToDerivedClassObj);
15   cout << ptrToBaseClassObj->toString() <<endl;
16 }
[motoki@x205a]$ g++ testStoringPtrToDerivObjInVirtualBaseVar.cpp
   ColoredRectangle_verVirtualToString.cpp Rectangle_verVirtualToString.cpp
[motoki@x205a]$ ./a.out
rectangle(id=0,width=1,height=2,color="black")
rectangle(id=0,width=1,height=2,color="black")
[motoki@x205a]$
```

この実行結果に関して、

◇ プログラム 15 行目 の実行により、クラス Rectangle のメンバ関数 toString() ではなく、ポインタ ptrToBaseClassObj がその時に指しているオブジェクトの本当のデータ型である、ColoredRectangle のメンバ関数 toString() が実行されていることが確認される。

5.4 抽象クラス

{Pohl(1999)8.4 節,8.7 節前書き,8.7.1 節, 柴田 (2014)6.1-6.2 節 プログラミング I(2017) 例 22.6, 例 22.11}

基底クラス内の仮想 (メンバ) 関数については、

virtual データ型 funcName(仮引数列) = 0;

という形で宣言することによって、関数本体の処理を未定義のままでクラス定義を終えることができる。C++言語では、この様に本体処理が未定義のままの仮想関数を**純粋仮想関数** (pure virtual function) と呼ぶ。また、純粋仮想 (メンバ) 関数を 1 個以上含むクラスを**抽象クラス**と呼ぶ。これに関して、

- 当然のことながら、抽象クラスに属するオブジェクトを生成することは出来ない。
- 抽象クラス型へのポインタ変数を用意して、派生クラスのインスタンスへのポインタを保持させることはできる。

抽象クラスを使う利点：

- 派生クラスの構成の仕方についての基本的な枠組みを設定することになり、(多人数で仕事を分担する場合でも) 統一感のあるプログラムの作成に繋がる。
- (派生クラス側に抽象メソッドの記述を要求することになるので、) メンバ関数のオーバーライドのやり忘れをコンパイルの時点で防げる。
- (インスタンスを生成できないので、) 意図しないインスタンス生成をコンパイルの時点で防げる。
- 互いに類似した複数のクラスがある場合、
  ◇ これらのクラスに共通する部分の詳細だけを記述し、

◇ (必要性についてはこれらの類似クラスに共通だが処理の中身は) 個別対応が必要なメンバ関数については、中身の記述は避けて純粋仮想関数にした

抽象クラスを用意して、元々必要なクラスをこの抽象クラスの派生クラスとして定義することにすれば、

◇ 同じコードをあちこちの場所に書く必要が無くなり、コードの冗長さが無くなる。
◇ 各派生クラスには派生クラス特有の処理だけを書けばよく、コードの見通しが良くなる。
◇ 別の類似クラスが新たに必要になった場合も、そのクラスを簡単に定義できる様になる。

例 5.7 (2 次元座標上の図形オブジェクトに共通の枠組みを定める抽象クラス) 2次元座標上の円や長方形、三角形といった図形オブジェクトを多数扱う場合、これらに共通の枠組みとして抽象クラスを考え、個々の種類のオブジェクトのクラスをこの抽象クラスの派生クラスとして定義することにすれば、図形オブジェクト全体を統一的に扱うことができる様になる。具体例として、図形オブジェクト全体の抽象基底クラス Shape2D と円オブジェクトの派生クラス Circle2D, 長方形オブジェクトの派生クラス Rectangle2D, 三角形オブジェクトの派生クラス Triangle2D の定義例を次に示す。

```
[motoki@x205a]$ cat -n Shape2D.h
     1 /* 頂点の座標情報等を保持する 2 次元図形オブジェクト          */
     2 /* に共通の枠組みを定める抽象基底クラス Shape2D （仕様部）*/
     3
     4 #ifndef ___Class_Shape2D
     5 #define ___Class_Shape2D
     6
     7 #include <string>
     8
     9 class Shape2D {
    10   static int numOfInstances;  // これまでに生成したインスタンスの個数
    11 protected:
    12   const int id;    // 図形インスタンスに付ける id 番号
    13   Shape2D(): id(numOfInstances++) {}
    14 public:
    15   int getId() const { return id; }
    16   virtual std::string toString() const = 0;
                                  //内部保持の図形情報を string データとして返す
    17   virtual double getArea() const = 0;
    18 };
    19
    20 #endif
[motoki@x205a]$ cat -n Shape2D.cpp
     1 /* 頂点の座標情報等を保持する 2 次元図形オブジェクト          */
     2 /* に共通の枠組みを定める抽象基底クラス Shape2D （実装部）*/
     3
     4 #include "Shape2D.h"
     5
     6 // static 変数の初期化 ------------------------------------------
     7 int Shape2D::numOfInstances = 0;
[motoki@x205a]$ cat -n Circle2D.h
```

```
 1 /* 中心座標と半径の情報を保持する 2 次元円オブジェクト */
 2 /* のクラス Circle2D                          (仕様部) */
 3
 4 #ifndef ___Class_Circle2D
 5 #define ___Class_Circle2D
 6
 7 #include <string>
 8 #include "Shape2D.h"
 9
10 const double PI = 3.1415926535897932;  //円周率
11
12 class Circle2D : public Shape2D {
13   double x;       //円の中心の x 座標
14   double y;       //円の中心の y 座標
15   double radius;  //円の半径
16 public:
17   Circle2D(double x=0.0, double y=0.0, double radius=1.0)
18     : Shape2D(), x(x), y(y), radius(radius) {}
19   std::string toString() const;
20   double getArea() const { return PI*radius*radius; }
21 };
22
23 #endif
[motoki@x205a]$ cat -n Circle2D.cpp
 1 /* 中心座標と半径の情報を保持する 2 次元円オブジェクト */
 2 /* のクラス Circle2D                          (実装部) */
 3
 4 #include <sstream>
 5 #include <string>
 6 #include "Circle2D.h"
 7 using namespace std;
 8
 9 // Circle2D 型オブジェクトを操作するための関数群 ----------------------
10 // オブジェクト内部に保持している円情報を string 型データとして返す
11 string Circle2D::toString() const
12 {
13   ostringstream ostr;
14   ostr << "circle[id=" << id
15        << "] of center (" << x << "," << y << ") and radius " << radius;
16   return ostr.str();
17 }
[motoki@x205a]$ cat -n Rectangle2D.h
 1 /* 頂点の座標情報を保持する 2 次元長方形オブジェクト */
 2 /* のクラス Rectangle2D                       (仕様部) */
 3
 4 #ifndef ___Class_Rectangle2D
 5 #define ___Class_Rectangle2D
 6
 7 #include <string>
 8 #include <cmath>
 9 #include "Shape2D.h"
10
```

```
    11 class Rectangle2D : public Shape2D {
    12   double x0, y0;       //長方形の 1 つの頂点の x 座標,y 座標
    13   double x1, y1;       //(x0,y0) と対角の位置にある頂点の x 座標,y 座標
    14 public:
    15   Rectangle2D(double x0=0.0, double y0=0.0, double x1=1.0, double y1=1.0)
    16     : Shape2D(), x0(x0), y0(y0), x1(x1), y1(y1) {}
    17   std::string toString() const;
    18   double getArea() const { return fabs((x1-x0)*(y1-y0)); }
    19 };
    20
    21 #endif
[motoki@x205a]$ cat -n Rectangle2D.cpp
     1 /* 頂点の座標情報を保持する 2 次元長方形オブジェクト */
     2 /* のクラス Rectangle2D                    (実装部) */
     3
     4 #include <sstream>
     5 #include <string>
     6 #include "Rectangle2D.h"
     7 using namespace std;
     8
     9 // Rectangle2D 型オブジェクトを操作するための関数群 ----------------------
    10 // オブジェクト内部に保持している長方形情報を string 型データとして返す
    11 string Rectangle2D::toString() const
    12 {
    13   ostringstream ostr;
    14   ostr << "rectangle[id=" << id << "] of vertices ("
    15        << x0 << "," << y0 << "), ("
    16        << x1 << "," << y0 << "), ("
    17        << x1 << "," << y1 << "), ("
    18        << x0 << "," << y1 << ")";
    19   return ostr.str();
    20 }
[motoki@x205a]$ cat -n Triangle2D.h
     1 /* 頂点の座標情報を保持する 2 次元三角形オブジェクト */
     2 /* のクラス Triangle2D                    (仕様部) */
     3
     4 #ifndef ___Class_Triangle2D
     5 #define ___Class_Triangle2D
     6
     7 #include <string>
     8 #include "Shape2D.h"
     9
    10 class Triangle2D : public Shape2D {
    11   double x0, y0;       //三角形の 1 つ目の頂点の x 座標,y 座標
    12   double x1, y1;       //三角形の 2 つ目の頂点の x 座標,y 座標
    13   double x2, y2;       //三角形の 3 つ目の頂点の x 座標,y 座標
    14 public:
    15   Triangle2D(double x0=0.0, double y0=0.0,
    16       double x1=1.0, double y1=0.0, double x2=0.0, double y2=1.0)
    17     : Shape2D(), x0(x0), y0(y0), x1(x1), y1(y1), x2(x2), y2(y2) {}
    18   std::string toString() const;
    19   double getArea() const;
```

```
    20 };
    21
    22 #endif
[motoki@x205a]$ cat -n Triangle2D.cpp
     1 /* 頂点の座標情報を保持する 2 次元三角形オブジェクト */
     2 /* のクラス Triangle2D                    (実装部) */
     3
     4 #include <sstream>
     5 #include <string>
     6 #include <cmath>
     7 #include "Triangle2D.h"
     8 using namespace std;
     9
    10 // Rectangle2D 型オブジェクトを操作するための関数群 ----------------------
    11 // オブジェクト内部に保持している三角形情報を string 型データとして返す
    12 string Triangle2D::toString() const
    13 {
    14   ostringstream ostr;
    15   ostr << "triangle[id=" << id << "] of vertices ("
    16        << x0 << "," << y0 << "), ("
    17        << x1 << "," << y1 << "), ("
    18        << x2 << "," << y2 << ")";
    19   return ostr.str();
    20 }
    21
    22 // オブジェクト内部に保持している 2 次元座標上の三角形の面積を計算して返す
    23 double Triangle2D::getArea() const
    24 {                                                    //ヘロンの公式
    25   double sideLeng1 = sqrt((x0-x1)*(x0-x1)+(y0-y1)*(y0-y1));
    26   double sideLeng2 = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
    27   double sideLeng3 = sqrt((x2-x0)*(x2-x0)+(y2-y0)*(y2-y0));
    28   double s = (sideLeng1+sideLeng2+sideLeng3)/2.0;
    29   return sqrt(s*(s-sideLeng1)*(s-sideLeng2)*(s-sideLeng3));
    30 }
[motoki@x205a]$
```

これに関して、

- Shape2D.h の 10 行目、12 行目、13 行目中の"id(numOfInstances++)"の部分、Shape2D.cpp の 7 行目 は、個々の図形インスタンスに固有の id 番号を割り振るためのコードである。Shape2D.h,12 行目では、インスタンス内に id 番号を入れるためのデータ領域が確保され、派生クラスのインスタンスから自由にアクセスできる様に (private ではなく)protected 指定され、一旦割り当てた id 番号が後で変更されない様に const と宣言されている。生成される図形インスタンスに順に 1, 2, 3, ... という id 番号を割り振るために、Shape2D.h,10 行目および Shape2D.cpp,7 行目では 過去に生成した図形インスタンスの個数を保持する static 変数を宣言し、Shape2D.h,13 行目では 派生クラスのコンストラクタが呼び出された際にインスタンスに id 番号を割り振る作業をすることを示している。
- Shape2D.h の 16~17 行目 で派生クラスのインスタンスの備えるべき関数名が規定されているので、派生クラスの定義はこの規定に沿ったものになり、全体的な統一感がある程度保証されることになる。

- Triangle2D.cpp の 23~30 行目においては、三角形の面積を計算するのにヘロンの公式、すなわち

  > 三角形の 3 辺の長さを a, b, c としたとき、
  > 面積 $= \sqrt{s(s-a)(s-b)(s-c)}$, 但し $s = (a+b+c)/2$

  という公式を用いている。

- 上の様にクラス Shape2D, Circle2D, Rectangle2D, Triangle2D が定義されていれば、それらを使って次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n useDerivedClassesFromShape2D.cpp
     1 /* Circle2D.h, Circle2D.cpp,              */
     2 /* Rectangle2D.h, Rectangle.cpp,          */
     3 /* Triangle2D.h, Triangle2D.cpp の利用例 */
     4
     5 #include <iostream>
     6 #include "Circle2D.h"
     7 #include "Rectangle2D.h"
     8 #include "Triangle2D.h"
     9 using namespace std;
    10
    11 int main()
    12 {
    13   Shape2D* fig = new Circle2D(1.0, 0.0, 2.0);  // fig... 多態変数
    14   cout << "fig = " << fig->toString() << endl
    15        << "  ==> fig->getArea() = " << fig->getArea() << endl;
    16   delete fig;
    17
    18   fig = new Rectangle2D(0.0, 0.0, 1.0, 2.0);
    19   cout << "fig = " << fig->toString() << endl
    20        << "  ==> fig->getArea() = " << fig->getArea() << endl;
    21   delete fig;
    22
    23   fig = new Triangle2D(0.0, 0.0, 2.0, 0.0, 1.0, 1.0);
    24   cout << "fig = " << fig->toString() << endl
    25        << "  ==> fig->getArea() = " << fig->getArea() << endl;
    26   delete fig;
    27 }
[motoki@x205a]$ g++ useDerivedClassesFromShape2D.cpp Circle2D.cpp
                          Rectangle2D.cpp Triangle2D.cpp Shape2D.cpp
[motoki@x205a]$ ./a.out
fig = circle[id=0] of center (1,0) and radius 2
  ==> fig->getArea() = 12.5664
fig = rectangle[id=1] of vertices (0,0), (1,0), (1,2), (0,2)
  ==> fig->getArea() = 2
fig = triangle[id=2] of vertices (0,0), (2,0), (1,1)
  ==> fig->getArea() = 1
[motoki@x205a]$
```

5.5 Makefile を用いた分割コンパイル

C++言語では、通常、定義するクラス毎に.h ファイルと.cpp ファイルの 2 つを用意するので、大量の個数のソースファイルを把握した上でのプログラミング作業になることも多い。こんな時は、Makefile を利用することもできる。

5.5.1 heapsort vs. bubblesort vs. llistsort

{ プログラミング AI(2018) 例 14.6}

例題 5.8 (整列化モジュールに共通の枠組みを定める抽象クラス) プログラミング AI 例題 14.4(C 言語) で行った、

> 3 つの整列化モジュール btree-heapsort.c, bubblesort.c, llistsort.c の外部仕様 (i.e. 外向けに提供する外部関数の名前や使い方) の統一

に相当することを C++の抽象クラスの考え方の下で行ってみよ。

(考え方) 例 5.7 に倣って、

> int 配列内の要素を昇順に並べ替える機能を備えた整列化モジュール

に共通の枠組みとして抽象クラスを考え、個々の整列化手法ごとに、その手法で整列化するモジュールのクラスを抽象クラスの派生クラスとして定義すればよい。より具体的には、

- 抽象クラスの設計に関して :
  プログラミング AI 例題 14.4 では整列化モジュール内で定義する関数仕様を次の様に統一した。
  - ◇ `void sort_method(char *method)` ··· 引数として指定された char 型配列に 整列化手法の名前を表す文字列を入れる関数,
  - ◇ `void sort(int a[], int size)` ··· int 型配列の名前 a[] とその大きさ size を引数として受け取り、指定された配列内のデータを各々の整列化手法で小さい順に並べ替える関数

  この 2 つに相当するインタフェースを目的とする抽象クラスに設定すれば良い。まず、関数 sort() については、そのインタフェースを整列化モジュールのインタフェースとして用いることに何の違和感もないので、単に、抽象クラスの定義の中に、

  ```
  virtual void sort(int a[], const int size) const = 0;
  ```

  という純仮想メンバ関数の宣言を入れて、利用の際のインタフェース統一を図れば十分である。次に、関数 sort_method() については、関数呼び出しの際に十分な容量を持った char 型配列の名前を実引数に与える必要があり、使用間違いの可能性もある。そこで、ここでは sort_method() と同等のメンバ関数を導入するのは避け、代わりに、

  ```
  virtual std::string toString() const = 0;
  ```

  という形の純仮想メンバ関数を導入して、利用の際のインタフェース統一を図ることにする。(このメンバ関数には、それぞれのモジュール (オブジェクト) の説明として整列化手法等も答えてもらうことを期待する。)

- 派生クラスの記述に関して：
  3 つの整列化手法については C プログラム btree-heapsort.c(プログラミング AI 例題 13.2), bubblesort.c(プログラミング AI 例題 14.4), llistsort.c(プログラミング AI 例題 14.4) に記述されている通りで、大部分の記述が、各々の C プログラム内で定義されている sort() 関数を派生クラス内のメンバ関数として取り込むだけで終わる。線形リストの実装については、例題 4.5 を参考に作れば良い。

(プログラミング) ここで関連するクラスとして、整列化モジュール全体の抽象クラス SortModuleForIntArray, heapsort モジュールの派生クラス HeapsortIntArray, bubblesort モジュールの派生クラス BubblesortIntArray, 連結リストへの挿入に基づく整列化モジュールの派生クラス LListsortIntArray, int データを各節点に保持する連結リストオブジェクトのクラス SortedLinkedListOfInt を定義した。得られた C++コードを次に示す。

```
[motoki@x205a]$ cat -n SortModuleForIntArray.h
     1 /* int 配列内の要素を昇順に並べ替える機能を備えた整列化モジュール        */
     2 /* に共通の枠組みを定める抽象基底クラス SortModuleForIntArray（仕様部）*/
     3
     4 #ifndef ___Class_SortModuleForIntArray
     5 #define ___Class_SortModuleForIntArray
     6
     7 #include <string>
     8
     9 class SortModuleForIntArray {
    10 public:
    11   // 整列化モジュールの説明 (主に手法) を答える
    12   virtual std::string toString() const = 0;
    13   // 引数で与えられた配列内の要素を昇順に並べ替える
    14   virtual void sort(int a[], const int size) const = 0;
    15 };
    16
    17 #endif
[motoki@x205a]$ cat -n HeapsortIntArray.h
     1 /* int 配列内の要素を heapsort 手法で昇順に並べ替える機能を備えた       */
     2 /* 整列化モジュールを作り出すためのクラス HeapsortIntArray（仕様部）*/
     3
     4 #ifndef ___Class_HeapsortIntArray
     5 #define ___Class_HeapsortIntArray
     6
     7 #include <string>
     8 #include "SortModuleForIntArray.h"
     9
    10 class HeapsortIntArray : public SortModuleForIntArray {
    11 public:
    12   // 整列化モジュールの説明 (主に手法) を答える
    13   std::string toString() const { return "Heapsort module"; }
    14   // 引数で与えられた配列内の要素を昇順に並べ替える
    15   void sort(int a[], const int size ) const;
    16 private:
    17   void heapify(int a[], const int treeSize, int hole, const int newElement) const;
    18 };
```

```
    19
    20 #endif
[motoki@x205a]$ cat -n HeapsortIntArray.cpp
     1 /* int 配列内の要素を heapsort 手法で昇順に並べ替える機能を備えた       */
     2 /* 整列化モジュールを作り出すためのクラス HeapsortIntArray（実装部）*/
     3
     4 #include "HeapsortIntArray.h"
     5
     6 // HeapsortIntArray 型オブジェクト内に用意された関数群 ===============
     7
     8 /*-------------------------------------------------------------*/
     9 /* 配列要素を小さい順に並べ替える (heapsort)                  */
    10 /*-----------------------------------------                    */
    11 /* (仮引数) a    : int 型配列                                  */
    12 /*          size : int 型配列 a の大きさ                       */
    13 /* (関数値)  :  なし                                          */
    14 /* (機能) :  heapsort アルゴリズムを使って、配列要素          */
    15 /*                a[0],a[1],a[2], ..., a[size-1]              */
    16 /*           を値の小さい順に並べ替える。                     */
    17 /*-------------------------------------------------------------*/
    18 void HeapsortIntArray::sort(int a[], const int size) const
    19 {
    20   //下から heap を構築してゆく
    21   for (int k=size/2-1; k>=0; --k)
    22     heapify(a, size, k, a[k]);
    23
    24   //大きい順に heap から取り出してゆく
    25   for (int k=size-1; k>=1; --k) {
    26       int tmp = a[k];
    27       a[k]    = a[0];
    28       heapify(a, k, 0, tmp);
    29   }
    30 }
    31
    32 /*-------------------------------------------------------------*/
    33 /* 番号 hole の節点より下の部分が heap の条件を満たす時、     */
    34 /* 新要素を加えて hole 以下の部分が heap の条件を満たす様にする。*/
    35 /*-----------------------------------------                    */
    36 /* (仮引数) a    : int 型配列                                  */
    37 /*     tree_size : 2 分木と見做す部分配列 a[0],a[1],... の大きさ*/
    38 /*          hole : a[0]〜a[tree_size] の表す 2 分木内の節点番号*/
    39 /*   new_element : 2 分木の節点に振り分けていない値           */
    40 /* (関数値)  :  なし                                          */
    41 /* (機能) :  番号 hole の節点のデータ記憶域は空で、その分    */
    42 /*           new_element という値がどの節点にも記録されてい */
    43 /*           ない、また、hole より下の部分が heap の条件を満  */
    44 /*           たしている、という状況を想定する。この様な状況 */
    45 /*           の時に、hole より下にあるデータを上に shift up */
    46 /*           する操作を繰り返し行い、適当な時点で空の節点に */
    47 /*           新しい要素 new_element を割り当てることにより、*/
    48 /*           hole 以下の部分が全面的に heap の条件を満たす様にする。*/
    49 /*-------------------------------------------------------------*/
```

```
    50 void HeapsortIntArray::heapify(int a[], const int tree_size,
    51        int hole, const int new_element) const
    52 {
    53   int  siftup_cand;      /* siftup candidate */
    54
    55   while ((siftup_cand = hole*2+1) < tree_size) {
    56     if (siftup_cand+1<tree_size                /*右の子も居て*/
    57         && a[siftup_cand]<a[siftup_cand+1])    /*右の子の方が*/
    58       ++siftup_cand;                           /*大きい場合は*/
    59                                          /*右の子が siftup の候補*/
    60     if ( new_element >= a[siftup_cand])
    61       break;       /* new_element を hole の場所に入れれば良い */
    62
    63     a[hole] = a[siftup_cand];     /* sift up */
    64     hole    = siftup_cand;
    65   }
    66   a[hole] = new_element;
    67 }
[motoki@x205a]$ cat -n BubblesortIntArray.h
     1 /* int 配列内の要素を bubblesort 手法で昇順に並べ替える機能を備えた       */
     2 /* 整列化モジュールを作り出すためのクラス BubblesortIntArray (仕様部) */
     3
     4 #ifndef ___Class_BubblesortIntArray
     5 #define ___Class_BubblesortIntArray
     6
     7 #include <string>
     8 #include "SortModuleForIntArray.h"
     9
    10 class BubblesortIntArray : public SortModuleForIntArray {
    11 public:
    12   // 整列化モジュールの説明 (主に手法) を答える
    13   std::string toString() const { return "Bubblesort module"; }
    14   // 引数で与えられた配列内の要素を昇順に並べ替える
    15   void sort(int a[], const int size ) const;
    16 };
    17
    18 #endif
[motoki@x205a]$ cat -n BubblesortIntArray.cpp
     1 /* int 配列内の要素を bubblesort 手法で昇順に並べ替える機能を備えた       */
     2 /* 整列化モジュールを作り出すためのクラス BubblesortIntArray (実装部) */
     3
     4 #include "BubblesortIntArray.h"
     5
     6 // BubblesortIntArray 型オブジェクト内に用意された関数群 ===============
     7
     8 /*-------------------------------------------------------*/
     9 /* 配列要素を小さい順に並べ替える (bubblesort)              */
    10 /*----------------------------------------                 */
    11 /* (仮引数) a    : int 型配列                               */
    12 /*          size : int 型配列 a の大きさ                    */
    13 /* (関数値) :  なし                                        */
    14 /* (機能) :  bubblesort アルゴリズムを使って、配列要素       */
```

```
    15 /*                a[0],a[1],a[2], ..., a[size-1]            */
    16 /*           を値の小さい順に並べ替える。                    */
    17 /*----------------------------------------------------------*/
    18 void BubblesortIntArray::sort(int a[], const int size) const
    19 {
    20   for (int i=0; i<size-1; ++i)
    21     for (int j=size-1; j > i; --j)
    22       if (a[j-1] > a[j]) {      /* a[j-1] と a[j]    */
    23         int temp = a[j-1];      /* の大小を調べて、   */
    24         a[j-1]   = a[j];        /* 逆順なら交換する。*/
    25         a[j]     = temp;
    26       }
    27 }
[motoki@x205a]$ cat -n LListsortIntArray.h
     1 /* (1)int配列内の要素を次々と連結リストの大小順を保つ位置に挿入して */
     2 /* いき、それが終わったら、(2) 連結リストに保持されたものを順に int 配 */
     3 /* 列内に移す、という手法で昇順に並べ替える機能を備えた             */
     4 /* 整列化モジュールを作り出すためのクラス LListsortIntArray (仕様部)*/
     5
     6 #ifndef ___Class_LListsortIntArray
     7 #define ___Class_LListsortIntArray
     8
     9 #include <string>
    10 #include "SortModuleForIntArray.h"
    11 #include "SortedLinkedListOfInt.h"
    12
    13 class LListsortIntArray : public SortModuleForIntArray {
    14 public:
    15   // 整列化モジュールの説明 (主に手法) を答える
    16   std::string toString() const
    17     { return "sort module that is based on insertion in a linked list"; }
    18   // 引数で与えられた配列内の要素を昇順に並べ替える
    19   void sort(int a[], const int size ) const;
    20 };
    21
    22 #endif
[motoki@x205a]$ cat -n LListsortIntArray.cpp
     1 /* (1)int配列内の要素を次々と連結リストの大小順を保つ位置に挿入して */
     2 /* いき、それが終わったら、(2) 連結リストに保持されたものを順に int 配 */
     3 /* 列内に移す、という手法で昇順に並べ替える機能を備えた             */
     4 /* 整列化モジュールを作り出すためのクラス LListsortIntArray (実装部)*/
     5
     6 #include "LListsortIntArray.h"
     7
     8 // LListsortIntArray 型オブジェクト内に用意された関数群 ===============
     9
    10 /*----------------------------------------------------------*/
    11 /* 配列要素を小さい順に並べ替える (線形リスト上での挿入繰返し) */
    12 /*------------------------------------------                */
    13 /* (仮引数) a    : int 型配列                               */
    14 /*          size : int 型配列 a の大きさ                    */
    15 /* (関数値)  :  なし                                       */
```

```
    16 /* (機能) :  配列要素                                          */
    17 /*                a[0],a[1],a[2], ..., a[size-1]              */
    18 /*           線形リスト上に昇順に挿入していき、その結果を    */
    19 /*           配列に戻すことによって小さい順に並べ替える      */
    20 /*-------------------------------------------------------------*/
    21 void LListsortIntArray::sort(int a[], const int size) const
    22 {
    23   SortedLinkedListOfInt list;
    24
    25   for (int i=0; i<size; ++i)
    26     list.insertNodeOf(a[i]);
    27
    28   list.moveAllNodeInfoTo(a);
    29 }
[motoki@x205a]$ cat -n SortedLinkedListOfInt.h
     1 /* 整数を要素にもつ Node 群を整数の小さい順に連結した、 */
     2 /* 連結リストのクラス SortedLinkedListOfInt (仕様部)  */
     3
     4 #ifndef ___Class_SortedLinkedListOfInt
     5 #define ___Class_SortedLinkedListOfInt
     6
     7 #include <cstddef>   // for NULL
     8
     9 struct NodeInt {
    10   const int num;
    11   NodeInt* next;
    12   NodeInt(int num=0, NodeInt* next=NULL): num(num), next(next) {}
    13 };
    14
    15 class SortedLinkedListOfInt {
    16   NodeInt* head;
    17 public:
    18   SortedLinkedListOfInt(): head(NULL) {}
    19   ~SortedLinkedListOfInt() { removeAllNodes(); }
    20   int getHeadNodeInfo() const;          //先頭 Node 内の情報を返す
    21   int getNumOfNodesOf(const int num) const;//引数指定の Node 数を返す
    22   void printAllNodeInfo() const;              //全ての Node 内の情報を出力
    23   void moveAllNodeInfoTo(int a[]);
                                  //全ての Node 内の int 値を引数指定の配列に移動
    24   void insertNodeOf(const int num);           //新データ挿入
    25   int removeHeadNode();                       //先頭 Node の削除
    26   int remove1stNodeOf(const int num); //引数指定の最初の Node を削除
    27   int removeAllNodesOf(const int num);//引数指定の全 Node を削除
    28   void removeAllNodes();                      //全 Node を削除
    29 };
    30
    31 #endif
[motoki@x205a]$ cat -n SortedLinkedListOfInt.cpp
     1 /* 整数を要素にもつ Node 群を整数の小さい順に連結した、 */
     2 /* 連結リストのクラス SortedLinkedListOfInt (実装部)  */
     3
     4 #include <iostream>
```

```
 5 #include <iomanip>
 6 #include <cstddef>   // for NULL
 7 #include "SortedLinkedListOfInt.h"
 8 using namespace std;
 9
10 // 連結リストの先頭 Node の num 値を返す
11 int SortedLinkedListOfInt::getHeadNodeInfo() const {
12   return head->num;
13 }
14
15 // 連結リスト中で引数を num 要素とする Node の個数を返す
16 int SortedLinkedListOfInt::getNumOfNodesOf(const int num) const {
17   NodeInt* ptr = head;
18   while (ptr != NULL && ptr->num < num)
19     ptr = ptr->next;
20   int count;
21   for (count=0; ptr != NULL && ptr->num == num; ++count)
22     ptr = ptr->next;
23   return count;
24 }
25
26 // 連結リスト内に保持されている内容を表の形に出力
27 void SortedLinkedListOfInt::printAllNodeInfo() const {
28   cout << "番号      num" << endl
29        << "----  ----------" << endl;
30   int count = 0;
31   for (NodeInt* ptr=head; ptr != NULL; ptr = ptr->next) {
32     cout << setw(4) << ++count
33  << setw(12) << ptr->num << endl;
34   }
35 }
36
37 // 連結リスト内の全 int 値を引数指定の配列に移動 (連結リストは空にする)
38 void SortedLinkedListOfInt::moveAllNodeInfoTo(int a[]) {
39   for (int n=0; head != NULL; ++n) {
40     a[n] = head->num;
41     NodeInt* tmp = head;
42     head = head->next;
43     delete tmp;
44   }
45 }
46
47 // 引数要素をもつ Node を連結リスト内の num 値の小さい順を保つ位置に挿入
48 void SortedLinkedListOfInt::insertNodeOf(const int num) {
49   NodeInt** ptrptr = &head;
50   while ((*ptrptr)!=NULL && (*ptrptr)->num <= num)
51     ptrptr = &((*ptrptr)->next);
52   *ptrptr = new NodeInt(num, *ptrptr);
53 }
54
55 // 連結リスト中の先頭 Node を削除し、削除した Node 数を返す
56 int SortedLinkedListOfInt::removeHeadNode() {
```

```
    57   if (head == NULL)
    58     return 0;
    59   NodeInt* tmp = head;
    60   head = head->next;
    61   delete tmp;
    62   return 1;
    63 }
    64
    65 // 連結リスト中で引数を num 要素とする最初の Node 削除し、削除した Node 数を返す
    66 int SortedLinkedListOfInt::remove1stNodeOf(const int num) {
    67   NodeInt** ptrptr = &head;
    68   while (*ptrptr != NULL && (*ptrptr)->num < num)
    69     ptrptr = &((*ptrptr)->next);
    70   if (*ptrptr != NULL && (*ptrptr)->num == num) {
    71     NodeInt* tmp = *ptrptr;
    72     *ptrptr = (*ptrptr)->next;
    73     delete tmp;
    74     return 1;
    75   }else {
    76     return 0;
    77   }
    78 }
    79
    80 // 連結リスト中で引数を num 要素とする Node を全て削除し、削除した Node 数を返す
    81 int SortedLinkedListOfInt::removeAllNodesOf(const int num) {
    82   NodeInt** ptrptr = &head;
    83   while (*ptrptr != NULL && (*ptrptr)->num < num)
    84     ptrptr = &((*ptrptr)->next);
    85   int count = 0;
    86   for (count=0; *ptrptr != NULL && (*ptrptr)->num == num; ++count) {
    87     NodeInt* tmp = *ptrptr;
    88     *ptrptr = (*ptrptr)->next;
    89     delete tmp;
    90   }
    91   return count;
    92 }
    93
    94 // 連結リスト中の Node を全て削除
    95 void SortedLinkedListOfInt::removeAllNodes() {
    96   while (head != NULL) {
    97     NodeInt* tmp = head;
    98     head = head->next;
    99     delete tmp;
   100   }
   101 }
[motoki@x205a]$
```

これに関して、

- 上記のプログラミング・デバッグの際に利用した Makefile(ここで関連した部分のみ), main() 関数を含むテスト実行用のコード、およびコンパイル・実行の様子を次に示す。

```
[motoki@x205a]$ cat -n Makefile
     1 # Makefile for clocking or testing 3 sorting methods:
```

```
 2 # (1) Heapsort
 3 # (2) Bubblesort
 4 # (3) Sort by insertion over linked-list
 5
 6 CC    = g++
      (途中省略)
36 SRCOBJS_TEST_HEAP_LIGHT  = testHeapsortIntArray_light.cpp \
37                     HeapsortIntArray.o
38 SRCOBJS_TEST_BUBBL_LIGHT = testBubblesortIntArray_light.cpp \
39                     BubblesortIntArray.o
40 SRCOBJS_TEST_LLIST_LIGHT = testLListsortIntArray_light.cpp \
41                     LListsortIntArray.o SortedLinkedListOfInt.o
      (途中省略)
63 test_heap_light: ${SRCOBJS_TEST_HEAP_LIGHT}
64 [tab] ${CC} -o test_heap_light $SRCOBJS_TEST_HEAP_LIGHT
65 test_bubble_light: ${SRCOBJS_TEST_BUBBL_LIGHT}
66 [tab] ${CC} -o test_bubble_light $SRCOBJS_TEST_BUBBL_LIGHT
67 test_llist_light: ${SRCOBJS_TEST_LLIST_LIGHT}
68 [tab] ${CC} -o test_llist_light $SRCOBJS_TEST_LLIST_LIGHT
      (途中省略)
77 #---------------------------------------------------------------------
      (途中省略)
85 HeapsortIntArray.o: HeapsortIntArray.cpp HeapsortIntArray.h \
                                                SortModuleForIntArray.h
86 [tab] ${CC} -c HeapsortIntArray.cpp
87 BubblesortIntArray.o: BubblesortIntArray.cpp BubblesortIntArray.h \
                                                SortModuleForIntArray.h
88 [tab] ${CC} -c BubblesortIntArray.cpp
89 LListsortIntArray.o: LListsortIntArray.cpp LListsortIntArray.h \
                         SortModuleForIntArray.h SortedLinkedListOfInt.h
90 [tab] ${CC} -c LListsortIntArray.cpp
91 SortedLinkedListOfInt.o: SortedLinkedListOfInt.cpp \
                                                SortedLinkedListOfInt.h
92 [tab] ${CC} -c SortedLinkedListOfInt.cpp
93
94 #---------------------------------------------------------------------
95 clean:
96 [tab] for i in *.o clock_all clock_heap clock_bubble clock_llist \
97 [tab]              test_all test_heap test_bubble test_llist \
98 [tab]              test_all_light test_heap_light \
99 [tab]              test_bubble_light test_llist_light ; do \
100 [tab]   if [ -f $$i ] ; then rm $$i ; fi \
101 [tab] done
[motoki@x205a]$ cat -n testHeapsortIntArray_light.cpp
 1 /* HeapsortIntArray.h, HeapsortIntArray.cpp の利用例 */
 2
 3 #include <iostream>
 4 #include "HeapsortIntArray.h"
 5 using namespace std;
 6
 7 int main()
 8 {
```

```
     9   int a[10] = {9, 8, 6, 7, 5, 3, 1, 2, 4, 0};
    10   SortModuleForIntArray* ptrSortModule = new HeapsortIntArray();
    11
    12   ptrSortModule->sort(a, 10);
    13   cout << "after sorting (" << ptrSortModule->toString() << ")" << endl;
    14   cout << "  a = {";
    15   for (int i=0; i<9; ++i)
    16     cout << a[i] << ", ";
    17   cout << a[9] << "}" << endl;
    18 }
[motoki@x205a]$ make test_heap_light
g++ -c HeapsortIntArray.cpp
g++ -o test_heap_light testHeapsortIntArray_light.cpp HeapsortIntArray.o
[motoki@x205a]$ ./test_heap_light
after sorting (Heapsort module)
  a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
[motoki@x205a]$ cat -n testBubblesortIntArray_light.cpp
     1 /* BubblesortIntArray.h, BubblesortIntArray.cpp の利用例 */
     2
     3 #include <iostream>
     4 #include "BubblesortIntArray.h"
     5 using namespace std;
     6
     7 int main()
     8 {
     9   int a[10] = {9, 8, 6, 7, 5, 3, 1, 2, 4, 0};
    10   SortModuleForIntArray* ptrSortModule = new BubblesortIntArray();
    11
    12   ptrSortModule->sort(a, 10);
    13   cout << "after sorting (" << ptrSortModule->toString() << ")" << endl;
    14   cout << "  a = {";
    15   for (int i=0; i<9; ++i)
    16     cout << a[i] << ", ";
    17   cout << a[9] << "}" << endl;
    18 }
[motoki@x205a]$ make test_bubble_light
g++ -c BubblesortIntArray.cpp
g++ -o test_bubble_light testBubblesortIntArray_light.cpp BubblesortIntArray.o
[motoki@x205a]$ ./test_bubble_light
after sorting (Bubblesort module)
  a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
[motoki@x205a]$ cat -n testLListsortIntArray_light.cpp
     1 /* LListsortIntArray.h, LListsortIntArray.cpp の利用例 */
     2
     3 #include <iostream>
     4 #include "LListsortIntArray.h"
     5 using namespace std;
     6
     7 int main()
     8 {
     9   int a[10] = {9, 8, 6, 7, 5, 3, 1, 2, 4, 0};
    10   SortModuleForIntArray* ptrSortModule = new LListsortIntArray();
```

```
    11
    12   ptrSortModule->sort(a, 10);
    13   cout << "after sorting (" << ptrSortModule->toString() << ")" << endl;
    14   cout << "  a = {";
    15   for (int i=0; i<9; ++i)
    16     cout << a[i] << ", ";
    17   cout << a[9] << "}" << endl;
    18 }
[motoki@x205a]$ make test_llist_light
g++ -c LListsortIntArray.cpp
g++ -c SortedLinkedListOfInt.cpp
g++ -o test_llist_light testLListsortIntArray_light.cpp
                                LListsortIntArray.o SortedLinkedListOfInt.o
[motoki@x205a]$ ./test_llist_light
after sorting (sort module that is based on insertion in a linked list)
  a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
[motoki@x205a]$
```

ここで、

◇ main() 関数を含むテスト用のコードに関しては、あちこちで利用するものでもない。また、依存するファイルも多数あり、例えば testHeapsortIntArray_light.o を生成するための Makefile の規則は暗黙のものでは不正確で、正確には

```
testHeapsortIntArray_light.o:  testHeapsortIntArray_light.cpp \
                        SortModuleForIntArray.h HeapsortIntArray.h
    tab  ${CC} -c testHeapsortIntArray_light.cpp
```

とでも定義しなければならない。こういう煩わしさを避けるため、上の Makefile,36 ∼41 行目 のマクロ定義では、main() を含むコードに関しては.o ファイルではなく.cpp ファイルの指定を行ない、.o ファイルの生成は行わないことにした。

例題 5.9 (整列化モジュールの動作テストを担当するモジュール) 例題 5.8 で定義した 3 つのクラス HeapsortIntArray, BubblesortIntArray, LListsortIntArray は共通の抽象基底クラスをもっている。従って、この基底クラス型へのポインタを通して多態的にメンバ関数の呼び出しを行うことにより、これらのインスタンス (整列化モジュール) を統合的に扱うことが出来る。これを体験するために、この種の整列化モジュールの提供する「int 配列内の要素を昇順に並べ替える機能」が正しく動作するかどうかをプログラミング AI 例題 13.2 の check-sort-program.c に倣ってテストする機能、すなわち

> ①0∼999 の間のランダムな整数を要素とする大きさ 100 の配列を生成し、
> ②それに対して与えられた整列化モジュールを適用して並べ替え作業を行い、
> ③その結果を出力する、

という風にテストする機能を備えたモジュールのクラスを定義してみよ。

(考え方) 共通の抽象基底クラスが SortModuleForIntArray であるので、この抽象クラスの型へのポインタ変数は多態変数 (に近いもの) になり、派生クラスのインスタンスへのポインタ値を保持することが出来る。従って、

> SortModuleForIntArray*型の変数を引数に持ち、引数で与えられた整列化モジュールに対して所定の動作テストを施すメンバ関数

を備えたモジュールのクラスを定義すれば良い。このメンバ関数の施す動作テストの処理内容については、C のコード check-sort-program.c を参考に C++風に書き直すだけである。

(プログラミング) 得られたクラス定義を次に示す。

```
[motoki@x205a]$ cat -n TesterForSortModuleIntArray.h
     1 /* SortModuleForIntArray モジュールの提供する                          */
     2 /* 「int 配列内の要素を昇順に並べ替える機能」が正しく動作するかどうか */
     3 /* をテストする機能を備えたモジュールを作り出すためのクラス（仕様部）*/
     4
     5 #ifndef ___Class_TesterForSortModuleIntArray
     6 #define ___Class_TesterForSortModuleIntArray
     7
     8 #include <string>
     9 #include "SortModuleForIntArray.h"
    10
    11 class TesterForSortModuleIntArray {
    12   static const int SIZE;   // runOnRandomData() の動作パラメータ
    13   static const int WIDTH;  // runOnRandomData() の動作パラメータ
    14 public:
    15   // オブジェクトの説明を string データとして返す
    16   std::string toString() const
    17     { return "Tester for module that is to sort int data in an array"; }
    18   // SIZE 個のランダムなデータから成る配列に対して
    19   // 引数で与えられた整列化モジュールを実行してみる
    20   void runOnRandomData(const SortModuleForIntArray* sortModule) const;
    21 private:
    22   // 引数で与えられた配列の要素を順に全て出力 (1 行に WIDTH 個ずつ)
    23   void prettyPrint(const int a[]) const;
    24 };
    25
    26 #endif
[motoki@x205a]$ cat -n TesterForSortModuleIntArray.cpp
     1 /* SortModuleForIntArray モジュールの提供する                          */
     2 /* 「int 配列内の要素を昇順に並べ替える機能」が正しく動作するかどうか */
     3 /* をテストする機能を備えたモジュールを作り出すためのクラス（実装部）*/
     4
     5 #include <iostream>
     6 #include <iomanip>
     7 #include <cstdlib>
     8 #include "TesterForSortModuleIntArray.h"
     9 using namespace std;
    10
    11 const int TesterForSortModuleIntArray::SIZE  = 100;
    12 const int TesterForSortModuleIntArray::WIDTH = 10;
    13
    14 // TesterForSortModuleIntArray 型オブジェクト内に用意された関数群 =========
    15
    16 // SIZE 個のランダムなデータから成る配列に対して
```

```
17 // 引数で与えられた整列化モジュールを実行してみる
18 void TesterForSortModuleIntArray::
19        runOnRandomData(const SortModuleForIntArray* sortModule) const
20 {
21   int a[SIZE], seed;
22
23   //擬似乱数の設定
24   cout << "擬似乱数の初期シード (int 値): ";
25   cin >> seed;
26   if (!cin) {
27     cout << "入力ミス --> 乱数シードの (再) 設定なし" << endl;
28   }else {
29     srand(seed);
30     cout << "乱数シードを" << seed << "に設定" << endl;
31   }
32
33   //配列 a の各々の要素に 0〜999 の乱数値を設定
34   for (int i=0; i<SIZE; ++i)
35     a[i] = rand() % 1000;
36
37   //整列化前の配列の内容を表示
38   cout << endl << "before sorting:" << endl;
39   prettyPrint(a);
40
41   //整列化
42   sortModule->sort(a, SIZE);
43
44   //整列化後の配列の内容を表示
45   cout << endl << "after sorting(" << sortModule->toString() << "):" << endl;
46   prettyPrint(a);
47 }
48
49 // 引数で与えられた配列の要素を順に全て出力 (1 行に WIDTH 個ずつ)
50 void TesterForSortModuleIntArray::prettyPrint(const int a[]) const
51 {
52   int NumOfEleInLine=0;
53
54   for (int i=0; i<SIZE; ++i) {
55     cout << setw(7) << a[i];
56     ++NumOfEleInLine;
57     if (NumOfEleInLine >= WIDTH) {
58       cout << endl;
59       NumOfEleInLine = 0;
60     }
61   }
62   if (NumOfEleInLine > 0)
63     cout << endl;
64 }
[motoki@x205a]$
```

これに関して、

- 上記のプログラミング・デバッグの際に利用した Makefile(ここで関連した部分のみ),

main() 関数を含むコード、およびコンパイル・実行の様子を次に示す。

```
[motoki@x205a]$ cat -n Makefile
     1 # Makefile for clocking or testing 3 sorting methods:
     2 # (1) Heapsort
     3 # (2) Bubblesort
     4 # (3) Sort by insertion over linked-list
     5
     6 CC    = g++
       (途中省略)
    22 SRCOBJS_TEST_ALL   = testSortModulesIntArray_random.cpp \
    23                     TesterForSortModuleIntArray.o \
    24                     HeapsortIntArray.o BubblesortIntArray.o \
    25                     LListsortIntArray.o SortedLinkedListOfInt.o
       (途中省略)
    52 test_all: ${SRCOBJS_TEST_ALL}
    53 [tab] ${CC} -o test_all ${SRCOBJS_TEST_ALL}
       (途中省略)
    77 #---------------------------------------------------------------------
       (途中省略)
    82 TesterForSortModuleIntArray.o: TesterForSortModuleIntArray.cpp \
                                                TesterForSortModuleIntArray.h
    83 [tab] ${CC} -c TesterForSortModuleIntArray.cpp
    84
    85 HeapsortIntArray.o: HeapsortIntArray.cpp HeapsortIntArray.h \
                                                    SortModuleForIntArray.h
    86 [tab] ${CC} -c HeapsortIntArray.cpp
    87 BubblesortIntArray.o: BubblesortIntArray.cpp BubblesortIntArray.h \
                                                    SortModuleForIntArray.h
    88 [tab] ${CC} -c BubblesortIntArray.cpp
    89 LListsortIntArray.o: LListsortIntArray.cpp LListsortIntArray.h \
                            SortModuleForIntArray.h SortedLinkedListOfInt.h
    90 [tab] ${CC} -c LListsortIntArray.cpp
    91 SortedLinkedListOfInt.o: SortedLinkedListOfInt.cpp SortedLinkedListOfInt.h
    92 [tab] ${CC} -c SortedLinkedListOfInt.cpp
    93
    94 #---------------------------------------------------------------------
    95 clean:
    96 [tab] for i in *.o clock_all clock_heap clock_bubble clock_llist \
    97 [tab]              test_all test_heap test_bubble test_llist \
    98 [tab]              test_all_light test_heap_light \
    99 [tab]              test_bubble_light test_llist_light ; do \
   100 [tab]   if [ -f $$i ] ; then rm $$i ; fi \
   101 [tab] done
[motoki@x205a]$ cat -n testSortModulesIntArray_random.cpp
     1 /* int 配列内の要素を昇順に並べ替える機能を備えた整列化モジュールとして */
     2 /* ・HeapsortIntArray オブジェクト,                                    */
     3 /* ・BubblesortIntArray オブジェクト,                                  */
     4 /* ・LListsortIntArray オブジェクト                                    */
     5 /* の 3 つを考え、これらが正しく整列化動作をするかどうかを             */
     6 /*       整列化モジュールをテストする機能を備えた                      */
     7 /*       TesterForSortModuleIntArray オブジェクト                      */
     8 /* を用いてテストする C++プログラム                                    */
```

```
 9
10
11 #include <iostream>
12 #include "HeapsortIntArray.h"
13 #include "BubblesortIntArray.h"
14 #include "LListsortIntArray.h"
15 #include "TesterForSortModuleIntArray.h"
16 using namespace std;
17
18 int main()
19 {
20   TesterForSortModuleIntArray tester;
21
22   //HeapsortIntArray オブジェクトの動作テスト
23   tester.runOnRandomData(new HeapsortIntArray());
24   cout << "---" << endl;
25
26   //BubblesortIntArray オブジェクトの動作テスト
27   tester.runOnRandomData(new BubblesortIntArray());
28   cout << "---" << endl;
29
30   //LListsortIntArray オブジェクトの動作テスト
31   tester.runOnRandomData(new LListsortIntArray());
32 }
[motoki@x205a]$ make test_all
g++ -c TesterForSortModuleIntArray.cpp
g++ -o test_all testSortModulesIntArray_random.cpp
      TesterForSortModuleIntArray.o HeapsortIntArray.o BubblesortIntArray.o
                                    LListsortIntArray.o SortedLinkedListOfInt.o
[motoki@x205a]$ ./test_all
擬似乱数の初期シード (int 値): 333
乱数シードを 333 に設定

before sorting:
    556    289    435    368    666    319    214    273    132    585
     64    943    869    956     50    298    112    218      5    649
    603    936    515    385    671    776    137    886      4    563
    718    913    204    153    281    870    473    495    144    605
    432    208    548    653    517    950    951    629    520    957
    630    476    893    498    861    917    626    998    803    631
    913    521    544    470     27    825    340    500    672    836
    105    104    397      6    110    914    308     61    895    829
     18    878    305    264    376    518    181    354    517    336
    985    782    857    881    252    236    706    945    736    730

after sorting(Heapsort module):
      4      5      6     18     27     50     61     64    104    105
    110    112    132    137    144    153    181    204    208    214
    218    236    252    264    273    281    289    298    305    308
    319    336    340    354    368    376    385    397    432    435
    470    473    476    495    498    500    515    517    517    518
    520    521    544    548    556    563    585    603    605    626
```

```
    629    630    631    649    653    666    671    672    706    718
    730    736    776    782    803    825    829    836    857    861
    869    870    878    881    886    893    895    913    913    914
    917    936    943    945    950    951    956    957    985    998
---
擬似乱数の初期シード (int 値): 333
乱数シードを 333 に設定

before sorting:
    556    289    435    368    666    319    214    273    132    585
     64    943    869    956     50    298    112    218      5    649
    603    936    515    385    671    776    137    886      4    563
    718    913    204    153    281    870    473    495    144    605
    432    208    548    653    517    950    951    629    520    957
    630    476    893    498    861    917    626    998    803    631
    913    521    544    470     27    825    340    500    672    836
    105    104    397      6    110    914    308     61    895    829
     18    878    305    264    376    518    181    354    517    336
    985    782    857    881    252    236    706    945    736    730

after sorting(Bubblesort module):
      4      5      6     18     27     50     61     64    104    105
    110    112    132    137    144    153    181    204    208    214
    218    236    252    264    273    281    289    298    305    308
    319    336    340    354    368    376    385    397    432    435
    470    473    476    495    498    500    515    517    517    518
    520    521    544    548    556    563    585    603    605    626
    629    630    631    649    653    666    671    672    706    718
    730    736    776    782    803    825    829    836    857    861
    869    870    878    881    886    893    895    913    913    914
    917    936    943    945    950    951    956    957    985    998
---
擬似乱数の初期シード (int 値): 333
乱数シードを 333 に設定

before sorting:
    556    289    435    368    666    319    214    273    132    585
     64    943    869    956     50    298    112    218      5    649
    603    936    515    385    671    776    137    886      4    563
    718    913    204    153    281    870    473    495    144    605
    432    208    548    653    517    950    951    629    520    957
    630    476    893    498    861    917    626    998    803    631
    913    521    544    470     27    825    340    500    672    836
    105    104    397      6    110    914    308     61    895    829
     18    878    305    264    376    518    181    354    517    336
    985    782    857    881    252    236    706    945    736    730

after sorting(sort module that is based on insertion in a linked list):
      4      5      6     18     27     50     61     64    104    105
    110    112    132    137    144    153    181    204    208    214
    218    236    252    264    273    281    289    298    305    308
    319    336    340    354    368    376    385    397    432    435
```

```
   470    473    476    495    498    500    515    517    517    518
   520    521    544    548    556    563    585    603    605    626
   629    630    631    649    653    666    671    672    706    718
   730    736    776    782    803    825    829    836    857    861
   869    870    878    881    886    893    895    913    913    914
   917    936    943    945    950    951    956    957    985    998
[motoki@x205a]$
```

ここで、

◇ 例題5.8 の場合と同じ理由で、上の Makefile,22~25 行目 のマクロ定義では、main() を含むコードに関しては.o ファイルではなく.cpp ファイルの指定を行ない、 .o ファイルの生成は行わないことにしている。

◇ プログラミング AI 例題 14.4(C 言語) で考えた 3 つの整列化モジュール btree-heapsort.c, bubblesort.c, llistsort.c については、関数名を含む関数仕様を同じにしてインタフェースを統一したために、同時に用いることが出来なかった。しかし、ここでの

```
void sort(int a[], const int size);
```

や

```
std::string toString();
```

は、クラスのメンバ関数なので、オブジェクト毎に別の関数として同時に用いることが出来る。そこで、ここでは 1 つのソースプログラム testSortModulesIntArray_random.cpp だけを用意し、その中で 3 種類の整列化モジュールの動作テストを行なっている。

例題 5.10 (時間計測モジュール, 整列化モジュールの動作速度を計測するモジュール)
プログラミング AI 例題 14.3 では、時間計測のためのモジュール consumed_time.c を C 言語で実装した。これに相当するオブジェクトのクラスを C++で実装せよ。更に、例題 5.8 で (抽象クラス SortModuleForIntArray から派生させて) 定義した 3 つのクラスのインスタンス (整列化モジュール) に対して、「int 配列内の要素を昇順に並べ替える手順」の動作速度をプログラミング AI 例題 14.4 の clock-sort-5-10-etc.c に倣って計測する機能、すなわち

> 要素数が 5, 10, 25, 50, 100, 200 の場合に対して
> ①問題例のランダムな設定,
> ②整列化プログラムの実行
> を各々 800000 回, 400000 回, 160000 回, 80000 回, 40000 回, 20000 回 繰り返して 1 回あたりの計算時間を求め、その結果を出力する、

という機能を備えたモジュールのクラスを定義してみよ。

(考え方) 時間計測モジュールに関しては、単に C 言語の consumed_time.c に相当するオブジェクトをインスタンスとするクラスを定義すれば良いだけである。ただ、C++では、定義する構造体名/クラス名には汎用性が求められ、またオブジェクト自体にも名前が付くので、構造体名／関数名を次の様に変更する。

C 言語		C++言語
データ型名 Second	⟶	構造体名 ProcessTime_realTime
関数 void start_timekeeper();	⟶	メンバ関数 void start();
関数 Second consumed_time();	⟶	メンバ関数 ProcessTime_realTime getLastIntervalTimes();

整列化モジュールの動作速度を計測するモジュールに関しては、例題5.9と同様に考えて、

> SortModuleForIntArray 型の変数を引数に持ち、引数で与えられた整列化モジュールに対して所定の方法で動作速度の計測を行うメンバ関数

を備えたモジュールのクラスを定義すれば良い。その際、動作速度の計測を行う手順については、C 言語で同様の処理を行っている clock-sort-5-10-etc.c (プログラミング AI 例題 14.4) を参考にすれば良い。

(プログラミング) 時間計測モジュールのクラス StopWatch, 整列化モジュールの動作速度を計測するモジュールのクラス TimerForSortModuleIntArray を次の様に定義した。

```
[motoki@x205a]$ cat -n StopWatch.h
     1 /* ストップウォッチ風に時間を測るためのオブジェクトのクラス StopWatch (仕様部) */
     2
     3 #ifndef ___Class_StopWatch
     4 #define ___Class_StopWatch
     5
     6 #include <string>
     7 #include <ctime>
     8
     9 struct ProcessTime_realTime {
    10   double processTime;
    11   double realTime;
    12 };
    13
    14 class StopWatch {
    15   clock_t previousClock;  //前回の clock() 値
    16   time_t  previousTime;   //前回の time(NULL) 値
    17   clock_t currentClock;   //現在の clock() 値
    18   time_t  currentTime;    //現在の time(NULL) 値
    19 public:
    20   StopWatch(): previousClock(-1), previousTime(-1.0),
    21     currentClock(-1), currentTime(-1.0) {}
    22   std::string toString() { return "module for measuring consumed time"; }
    23   //時間計測開始
    24   void start() ;
    25   //前回のマーク時点からの経過時間 (プロセス時間と実時間の組, 単位秒) を返す
    26   ProcessTime_realTime getLastIntervalTimes();
    27 };
    28
    29 #endif
[motoki@x205a]$ cat -n StopWatch.cpp
     1 /* ストップウォッチ風に時間を測るためのオブジェクトのクラス StopWatch (実装部) */
     2
     3 #include <cstddef>
     4 #include "StopWatch.h"
```

```
     5
     6 // StopWatch 型オブジェクト内に用意された関数群 ===============
     7
     8 //時間計測開始
     9 void StopWatch::start()
    10 {
    11   previousClock = clock();
    12   previousTime  = time(NULL);
    13 }
    14
    15 //前回のマーク時点からの経過時間 (プロセス時間と実時間の組, 単位秒) を返す
    16 ProcessTime_realTime StopWatch::getLastIntervalTimes()
    17 {
    18   ProcessTime_realTime intervalTimes;
    19
    20   currentClock = clock();
    21   currentTime  = time(NULL);
    22   intervalTimes.processTime
    23     = static_cast<double>(currentClock - previousClock) / CLOCKS_PER_SEC;
    24   intervalTimes.realTime = difftime(currentTime, previousTime);
    25   previousClock = currentClock;
    26   previousTime  = currentTime;
    27   return intervalTimes;
    28 }
[motoki@x205a]$ cat -n TimerForSortModuleIntArray.h
     1 /* SortModuleForIntArray モジュールの提供する                          */
     2 /* 「int 配列内の要素を昇順に並べ替える機能」の動作速度を計測する機能 */
     3 /* を備えたモジュールを作り出すためのクラス                (仕様部) */
     4
     5 #ifndef ___Class_TimerForSortModuleIntArray
     6 #define ___Class_TimerForSortModuleIntArray
     7
     8 #include <string>
     9 #include "SortModuleForIntArray.h"
    10
    11 //struct Problem {
    12 //  const int size;
    13 //  const int iterationNum;
    14 //};
    15
    16 class TimerForSortModuleIntArray {
    17 public:
    18   // オブジェクトの説明を string データとして返す
    19   std::string toString() const
    20     { return "Timer for module that is to sort int data in an array"; }
    21   // 要素数が 5, 10, 25, 50, 100, 200 の場合について、
    22   // 引数で与えられた整列化モジュールの平均実行時間を計る
    23   void clockingOnVariousProblems(const SortModuleForIntArray* sortModule) const;
    24 private:
    25   void setAnArrayRandom(int a[], const int size) const;
    26 };
    27
```

```
    28 #endif
[motoki@x205a]$ cat -n TimerForSortModuleIntArray.cpp
     1 /* SortModuleForIntArrayモジュールの提供する                              */
     2 /* 「int配列内の要素を昇順に並べ替える機能」の動作速度を計測する機能 */
     3 /* を備えたモジュールを作り出すためのクラス                  (実装部) */
     4
     5 #include <iostream>
     6 #include <iomanip>
     7 #include <cstdlib>
     8 #include <ctime>
     9 #include "StopWatch.h"
    10 #include "TimerForSortModuleIntArray.h"
    11 using namespace std;
    12
    13 // TesterForSortModuleIntArray型オブジェクト内に用意された関数群 =========
    14
    15 /*************************************************************/
    16 /* 要素数が 5, 10, 25, 50, 100, 200 の場合について、          */
    17 /* 整列化プログラムの平均実行時間を計る                      */
    18 /*-------------------------------------------                 */
    19 /*   要素数が 5, 10, 25, 50, 100, 200 の場合について、それ  */
    20 /*   ぞれ 800000回, 400000回, 160000回, 80000回, 40000回,    */
    21 /*   20000回 次の作業を繰り返して所用時間を計り、後で1回当り */
    22 /*   の計算時間を割り出す。                                   */
    23 /*       |配列上に要素数分だけランダムに整数を生成し、       */
    24 /*       |その配列要素を別途用意された整列化プログラムを     */
    25 /*       |使って昇順に並べ替える。                           */
    26 /*************************************************************/
    27 void TimerForSortModuleIntArray::
    28        clockingOnVariousProblems(const SortModuleForIntArray* sortModule) const
    29 {
    30   const int MAX_SIZE = 200;
    31   const int PROB_NUM = 6;
    32   struct Problem {
    33     const int size;
    34     const int iterationNum;
    35   };
    36   const Problem prob[PROB_NUM] = {{5,800000}, {10,400000}, {25,160000},
    37                                   {50,80000}, {100,40000}, {200,20000}};
    38   int a[MAX_SIZE], seed;
    39   StopWatch timer;
    40   ProcessTime_realTime timeForSort[PROB_NUM], timeForInit[PROB_NUM];
    41
    42   cout << "Clocking the average execution time of the program" << endl
    43        << "that sorts 5, 10, 25, 50, 100, or 200 elements." << endl
    44        << "  (*** " << sortModule->toString() << " ***)" << endl
    45        << "Input a random seed (0 - " << RAND_MAX << "):  ";
    46   cin >> seed;
    47
    48   //ソート以外の部分の計算時間を測定
    49   srand(seed);
    50   for (int k=0; k<PROB_NUM; ++k) {
```

```
51      timer.start();
52      for (int i=0; i<prob[k].iterationNum; ++i) { /* この部分の */
53        setAnArrayRandom(a, prob[k].size);         /* 計算時間を */
54      }                                          /* 参考のために測る。*/
55      timeForInit[k] = timer.getLastIntervalTimes();
56    }
57
58    //ソートプログラムの平均計算時間を測定
59    srand(seed);
60    for (int k=0; k<PROB_NUM; ++k) {
61      timer.start();
62      for (int i=0; i<prob[k].iterationNum; ++i) { /* この部分の */
63        setAnArrayRandom(a, prob[k].size);         /* 計算時間を */
64        sortModule->sort(a, prob[k].size);         /* 測る。      */
65      }                                          /*            */
66      timeForSort[k] = timer.getLastIntervalTimes();
67                       /* 次に sort 部分だけの計算時間を割り出す。 */
68      timeForSort[k].processTime -= timeForInit[k].processTime;
69      timeForSort[k].realTime    -= timeForInit[k].realTime;
70    }
71
72    // 測定結果の出力
73    cout << endl
74         << "        ** time for sort **    **time for initialize**" << endl
75         << "size    process_t  real_time    process_t  real_time" << endl
76         << "         (m sec)    (m sec)      (m sec)    (m sec)" << endl
77         << "----    ---------  ---------    ---------  ---------" << endl;
78    for (int k=0; k<PROB_NUM; ++k)
79      cout << setw(4) << prob[k].size << fixed << setprecision(5)
80           << "    "
81           << setw(9) << timeForSort[k].processTime*1000.0/prob[k].iterationNum
82           << "  "
83           << setw(9) << timeForSort[k].realTime*1000.0/prob[k].iterationNum
84           << "    "
85           << setw(9) << timeForInit[k].processTime*1000.0/prob[k].iterationNum
86           << "  "
87           << setw(9) << timeForInit[k].realTime*1000.0/prob[k].iterationNum
88           << endl;
89  }
90
91  /*-------------------------------------------------------------*/
92  /* 引数で与えられた配列の各要素をランダムに設定する          */
93  /*-----------------------------------------                    */
94  /* (仮引数) a    : int 型配列                                  */
95  /*          size : int 型配列 a の大きさ                       */
96  /* (関数値)  :  なし                                          */
97  /* (機能) :  配列要素 a[0]〜a[size-1] に 0〜999 の間の乱数  */
98  /*           を設定する。                                     */
99  /*-------------------------------------------------------------*/
100 void TimerForSortModuleIntArray::setAnArrayRandom(int a[], const int size) const
101 {
102   for (int i=0; i<size; ++i)
```

```
  103     a[i] = rand() % 1000;
  104 }
[motoki@x205a]$
```

これに関して、

- プログラミング AI 例題 14.4 の clock-sort-5-10-etc.c においては、Problem という名前のデータ型を

```
typedef struct {
   int  size;     /* 整列化する要素数      */
   int  ite_num;  /* 整列化の繰り返し回数 */
} Problem;
```

  という風に大域的な場所で定義したが、この構造体自体に汎用性は無いので、ここでは Problem 構造体をメンバ関数 TimerForSortModuleIntArray::clockingOnVariousProblems() 内で局所的に定義して用いている。

- 上記のプログラミング・デバッグの際に利用した Makefile(ここで関連した部分のみ), main() 関数を含むコード、およびコンパイル・実行の様子を次に示す。

```
[motoki@x205a]$ cat -n Makefile
     1 # Makefile for clocking or testing 3 sorting methods:
     2 # (1) Heapsort
     3 # (2) Bubblesort
     4 # (3) Sort by insertion over linked-list
     5
     6 CC    = g++
     7
     8 SRCOBJS_CLOCK_ALL   = clockSortModulesIntArray.cpp \
     9                       TimerForSortModuleIntArray.o StopWatch.o \
    10                       HeapsortIntArray.o BubblesortIntArray.o \
    11                       LListsortIntArray.o SortedLinkedListOfInt.o
        (途中省略)
    43 clock_all: ${SRCOBJS_CLOCK_ALL}
    44 [ tab ] ${CC} -o clock_all ${SRCOBJS_CLOCK_ALL}
        (途中省略)
    77 #-----------------------------------------------------------------------
    78 TimerForSortModuleIntArray.o: TimerForSortModuleIntArray.cpp \
                                                TimerForSortModuleIntArray.h
    79 [ tab ] ${CC} -c TimerForSortModuleIntArray.cpp
    80 StopWatch.o: StopWatch.cpp StopWatch.h
    81 [ tab ] ${CC} -c StopWatch.cpp
        (途中省略)
    85 HeapsortIntArray.o: HeapsortIntArray.cpp HeapsortIntArray.h \
                                                    SortModuleForIntArray.h
    86 [ tab ] ${CC} -c HeapsortIntArray.cpp
    87 BubblesortIntArray.o: BubblesortIntArray.cpp BubblesortIntArray.h \
                                                    SortModuleForIntArray.h
    88 [ tab ] ${CC} -c BubblesortIntArray.cpp
    89 LListsortIntArray.o: LListsortIntArray.cpp LListsortIntArray.h \
                           SortModuleForIntArray.h SortedLinkedListOfInt.h
    90 [ tab ] ${CC} -c LListsortIntArray.cpp
    91 SortedLinkedListOfInt.o: SortedLinkedListOfInt.cpp SortedLinkedListOfInt.h
    92 [ tab ] ${CC} -c SortedLinkedListOfInt.cpp
```

```
    93
    94 #---------------------------------------------------------------
    95 clean:
    96 [tab] for i in *.o clock_all clock_heap clock_bubble clock_llist \
    97 [tab]              test_all test_heap test_bubble test_llist \
    98 [tab]              test_all_light test_heap_light \
    99 [tab]              test_bubble_light test_llist_light ; do \
   100 [tab]   if [ -f $$i ] ; then rm $$i ; fi \
   101 [tab] done
[motoki@x205a]$ cat -n clockSortModulesIntArray.cpp
     1 /* int 配列内の要素を昇順に並べ替える機能を備えた整列化モジュールとして */
     2 /* ・HeapsortIntArray オブジェクト,                                    */
     3 /* ・BubblesortIntArray オブジェクト,                                  */
     4 /* ・LListsortIntArray オブジェクト                                    */
     5 /* の3つを考え、これら動作速度を                                       */
     6 /*       整列化モジュールの動作速度を計測する機能を備えた              */
     7 /*       TimerForSortModuleIntArray オブジェクト                       */
     8 /* を用いて調べる C++プログラム                                        */
     9
    10 #include <iostream>
    11 #include "HeapsortIntArray.h"
    12 #include "BubblesortIntArray.h"
    13 #include "LListsortIntArray.h"
    14 #include "TimerForSortModuleIntArray.h"
    15 using namespace std;
    16
    17 int main()
    18 {
    19   TimerForSortModuleIntArray timer;
    20
    21   //HeapsortIntArray オブジェクトの動作速度を計測
    22   timer.clockingOnVariousProblems(new HeapsortIntArray());
    23   cout << "---" << endl;
    24
    25   //BubblesortIntArray オブジェクトの動作速度を計測
    26   timer.clockingOnVariousProblems(new BubblesortIntArray());
    27   cout << "---" << endl;
    28
    29   //LListsortIntArray オブジェクトの動作速度を計測
    30   timer.clockingOnVariousProblems(new LListsortIntArray());
    31 }
[motoki@x205a]$ make clock_all
g++ -c TimerForSortModuleIntArray.cpp
g++ -c StopWatch.cpp
g++ -o clock_all clockSortModulesIntArray.cpp TimerForSortModuleIntArray.o
                        StopWatch.o HeapsortIntArray.o BubblesortIntArray.o
                              LListsortIntArray.o SortedLinkedListOfInt.o
[motoki@x205a]$ ./clock_all
Clocking the average execution time of the program
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** Heapsort module ***)
Input a random seed (0 - 2147483647):  333
```

```
        ** time for sort **    **time for initialize**
size    process_t  real_time    process_t  real_time
         (m sec)    (m sec)      (m sec)    (m sec)
----    ---------  ---------    ---------  ---------
   5      0.00007    0.00000      0.00008    0.00000
  10      0.00026    0.00250      0.00012    0.00000
  25      0.00104    0.00000      0.00029    0.00000
  50      0.00263    0.00000      0.00056    0.00000
 100      0.00634    0.00000      0.00111    0.00000
 200      0.01490    0.05000      0.00222    0.00000
---
Clocking the average execution time of the program
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** Bubblesort module ***)
Input a random seed (0 - 2147483647):  333

        ** time for sort **    **time for initialize**
size    process_t  real_time    process_t  real_time
         (m sec)    (m sec)      (m sec)    (m sec)
----    ---------  ---------    ---------  ---------
   5      0.00005    0.00125      0.00007    0.00000
  10      0.00027    0.00000      0.00012    0.00000
  25      0.00162    0.00000      0.00029    0.00000
  50      0.00595    0.01250      0.00059    0.00000
 100      0.02169    0.00000      0.00111    0.00000
 200      0.07558    0.10000      0.00222    0.00000
---
Clocking the average execution time of the program
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** sort module that is based on insertion in a linked list ***)
Input a random seed (0 - 2147483647):  333

        ** time for sort **    **time for initialize**
size    process_t  real_time    process_t  real_time
         (m sec)    (m sec)      (m sec)    (m sec)
----    ---------  ---------    ---------  ---------
   5      0.00022    0.00000      0.00007    0.00000
  10      0.00055    0.00250      0.00012    0.00000
  25      0.00170    0.00000      0.00029    0.00000
  50      0.00446    0.00000      0.00056    0.00000
 100      0.01321    0.02500      0.00111    0.00000
 200      0.04305    0.05000      0.00223    0.00000
[motoki@x205a]$
```

ここで、

◇ 例題5.8 や例題5.9 の場合と同じ理由で、上の Makefile,8~11 行目 のマクロ定義では、main() を含むコードに関しては.o ファイルではなく.cpp ファイルの指定を行ない、.o ファイルの生成は行わないことにしている。

◇ 例題5.9 の場合と同じ理由で、ここでは1つのソースプログラム clockSortModules IntArray.cpp だけを用意し、その中で3種類の整列化モジュールの動作速度計測を行なっている。

5.5.2 predator-prey シミュレーション

{Pohl(1999)8.4 節 }

例題 5.11 (predator-prey シミュレーション) 16×16 のマス目が 2 次元トーラス状に繋がり、その中の各々のマス目は①狐が 1 匹いるか、②兎が 1 匹いるか、③草が 1 株生えているか、④何も生息していないか、の状態にある。そして、次の単位時間後には各マスの状態は、自マスも含めた周囲 9 マスの状態に基づいて次の様に変化する。

- 現在、狐がいる時、

$$\text{次の状態} = \begin{cases} \text{空 (i.e. 生息なし)} & \text{if (周囲 9 マスの狐の数)} > 5 \\ \text{空 (i.e. 生息なし)} & \text{if (狐の年齢)} > 6 \\ \text{狐 (歳を重ねて継続)} & \text{otherwise} \end{cases}$$

- 現在、兎がいる時、

$$\text{次の状態} = \begin{cases} \text{空 (i.e. 生息なし)} & \text{if (周囲 9 マスの狐の数)}\geq\text{(周囲 9 マスの兎の数)} \\ \text{空 (i.e. 生息なし)} & \text{if (兎の年齢)} > 3 \\ \text{兎 (歳を重ねて継続)} & \text{otherwise} \end{cases}$$

- 現在、草が生息している時、

$$\text{次の状態} = \begin{cases} \text{空 (i.e. 生息なし)} & \text{if (周囲 9 マスの兎の数)}\geq\text{(周囲 9 マスの草の数)} \\ \text{草 (継続)} & \text{otherwise} \end{cases}$$

- 現在、何も生息していない時、

$$\text{次の状態} = \begin{cases} \text{狐 (0 歳)} & \text{if (周囲 9 マスの狐の数)} > 1 \\ \text{兎 (0 歳)} & \text{if (周囲 9 マスの狐の数)}\leq 1, \text{(周囲 9 マスの兎の数)} > 1 \\ \text{草 (新規)} & \text{if (周囲 9 マスの狐の数)}\leq 1, \text{(周囲 9 マスの兎の数)}\leq 1, \\ & \quad \text{(周囲 9 マスの草の数)} > 0 \\ \text{空 (i.e. 生息なし)} & \text{otherwise} \end{cases}$$

以上の状態遷移規則の下で各マスの状態が変化するとして、各時点での 16×16 のマス目の状態を観測する C++プログラムを作成せよ。

(考え方) シミュレーションの各時点で登場する、狐、兎、草、空 (i.e. 生息なし)、マス、16×16 のマス目全体、のそれぞれをプログラム上でオブジェクトとして扱うために、次の 7 つのクラスを用意する。

- `Cell` ... マス目のクラスで、次の主要メンバをもつ。
  - ◇ `LivingThing* life` ... マス目に生息する生物等へのポインタ
  - ◇ `Cell* ptrNeighborCell[3][3]` ... 周囲 9 マスへのポインタを要素とする 2 次元配列
- `LivingThing` ... 狐、兎、草、空 (i.e. 生息なし) を表すオブジェクト群を統一的に扱うための抽象クラスで、次の主要メンバをもつ。
  - ◇ `LifeType type` ... 生命の種類 (`LifeType` は列挙型)
  - ◇ `nextLife(Cell*)` ... 引数で与えられた周囲 9 マスへのポインタ情報を基に、次の単位時間後にマスに入る生物オブジェクト等へのポインタを返す純仮想関数
- `Fox` ... `LivingThing` から派生させた狐のクラスで、次の追加主要メンバをもつ。
  - ◇ `int age` ... 年齢
- `Rabbit` ... `LivingThing` から派生させた兎のクラスで、次の追加主要メンバをもつ。
  - ◇ `int age` ... 年齢
- `Grass` ... `LivingThing` から派生させた草のクラス。
- `NoLife` ... `LivingThing` から派生させた空 (i.e. 生息なし) のクラス。

- PredatorPreyWorld ... 16×16 のマス目全体を表すオブジェクトのクラスで、次の主要メンバをもつ。
  - ◇ Cell place[16][16] ... マス目を要素とする 2 次元配列
  - ◇ getNextLifeAtPlace(int, int) ... 引数で指定された座標のマス目に次の単位時間後に入る生物オブジェクト等へのポインタを返す関数
  - ◇ setLifeAtPlace(int, int , LivingThing*) ... 引数で指定された座標のマス目に引数の生物オブジェクト等へのポインタを組み込む関数
  - ◇ setLifeInformationEmpty() ... 全てのマス目内の生存生物情報を消去する関数
  - ◇ printConfiguration() ... 16×16 のマス目全体の生物生息の分布を出力する関数

そして、これらのクラスを利用して、16×16 のマス目の状態を連続的に状態遷移させるシミュレーションを行うコードを書けば良い。

(7 つのクラスの定義) 構成した 7 つのクラス Cell、LivingThing、Fox、Rabbit、Grass、NoLife、PredatorPreyWorld のコードを次に示す。

```
[motoki@x205a]$ cat -n Cell.h
     1 /* Predator-prey-simulation において                    */
     2 /* 仮想生物 1 個体の生存する場所のクラス Cell (仕様部) */
     3
     4 #ifndef ___Class_Cell
     5 #define ___Class_Cell
     6
     7 #include "LivingThing.h"
     8
     9 enum { SOUTH=0, WEST=0, NEUTRAL, NORTH, EAST=2 };
    10
    11 class Cell {
    12 public:
    13   LivingThing* life;              //             north
    14   Cell* ptrNeighborCell[3][3];    //      [2][0] [2][1] [2][2]
    15                                   // west [1][0]  自分  [1][2] east
    16                                   //      [0][0] [0][1] [0][2]
    17                                   //             south
    18   LivingThing* nextLife() { return life->nextLife(ptrNeighborCell); }
    19 };
    20
    21 #endif
[motoki@x205a]$ cat -n LivingThing.h
     1 /* 1 つの場所 (cell) 内に存在する仮想生物                        */
     2 /* に共通の枠組みを定める抽象基底クラス LivingThing (仕様部) */
     3
     4 #ifndef ___Class_LivingThing
     5 #define ___Class_LivingThing
     6
     7 //#include "Cell.h"    // #include "Cell.h" としても、
     8 class Cell;            //   LivingThing.h --(include)-> Cell.h
     9                        //     --(include, 回避)-> LivingThing.h
    10                        // という風にインクルードガードによって include
    11                        // の連鎖は回避され、クラス LivingThing が未定義
    12                        // の状態でクラス Cell の定義を行おうとしてエラー
```

```
    13                          // になるだけである。そこで、ここでは、左の様に、
    14                          // Cell がクラス名であることだけを (前方) 宣言する。
    15
    16 enum LifeType { NO_LIFE, GRASS, RABBIT, FOX };
    17 const int NUM_OF_POSSIBLE_LIFE_TYPES = 4;
    18
    19 class LivingThing {
    20 protected:
    21   LifeType type;
    22   LivingThing(LifeType type = NO_LIFE): type(type) {}
    23 public:
    24   LifeType getType() const { return type; }
    25   virtual LivingThing* nextLife(Cell* ptrNeighborCell[3][3]) = 0;
    26 };
    27
    28 #endif
[motoki@x205a]$ cat -n Fox.h
     1 /* Predator-prey-simulationにおいて、                                    */
     2 /* 1つの場所 (cell) 内に存在する「狐 (仮想生物)」のクラス Fox (仕様部) */
     3
     4 #ifndef ___Class_Fox
     5 #define ___Class_Fox
     6
     7 #include "LivingThing.h"
     8 #include "Cell.h"
     9
    10 const int FOX_LIFE_LENGTH = 6;
    11
    12 class Fox : public LivingThing {
    13 protected:
    14   int age;
    15 public:
    16   Fox(int age = 0): LivingThing(FOX), age(age) {}
    17   LivingThing* nextLife(Cell* ptrNeighborCell[3][3]);
    18 };
    19
    20 #endif
[motoki@x205a]$ cat -n Fox.cpp
     1 /* Predator-prey-simulationにおいて、                                    */
     2 /* 1つの場所 (cell) 内に存在する「狐 (仮想生物)」のクラス Fox (実装部) */
     3
     4 #include "Fox.h"
     5 #include "NoLife.h"
     6
     7 // Fox 型オブジェクトを操作するための関数群 --------------------------
     8 LivingThing* Fox::nextLife(Cell* ptrNeighborCell[3][3])
     9 {
    10   int numOf[NUM_OF_POSSIBLE_LIFE_TYPES];
    11
    12   numOf[NO_LIFE] = numOf[GRASS] = numOf[RABBIT] = numOf[FOX] = 0;
    13   for (int i=0; i<3; ++i) {
    14     for (int j=0; j<3; ++j) {
```

```
    15       ++numOf[ptrNeighborCell[i][j]->life->getType()];
    16     }
    17   }
    18
    19   if (numOf[FOX] > 5)               // too many foxes
    20     return (new NoLife());
    21   else if (age > FOX_LIFE_LENGTH)   // natural death
    22     return (new NoLife());
    23   else
    24     return (new Fox(age+1));
    25 }
[motoki@x205a]$ cat -n Rabbit.h
     1 /* Predator-prey-simulationにおいて、                                    */
     2 /* 1つの場所(cell)内に存在する「兎(仮想生物)」のクラス Fox (仕様部) */
     3
     4 #ifndef ___Class_Rabbit
     5 #define ___Class_Rabbit
     6
     7 #include "LivingThing.h"
     8 #include "Cell.h"
     9
    10 const int RABBIT_LIFE_LENGTH = 3;
    11
    12 class Rabbit : public LivingThing {
    13 protected:
    14   int age;
    15 public:
    16   Rabbit(int age = 0): LivingThing(RABBIT), age(age) {}
    17   LivingThing* nextLife(Cell* ptrNeighborCell[3][3]);
    18 };
    19
    20 #endif
[motoki@x205a]$ cat -n Rabbit.cpp
     1 /* Predator-prey-simulationにおいて、                                    */
     2 /* 1つの場所(cell)内に存在する「兎(仮想生物)」のクラス Fox (実装部) */
     3
     4 #include "Rabbit.h"
     5 #include "NoLife.h"
     6
     7 // Rabbit型オブジェクトを操作するための関数群 --------------------------
     8 LivingThing* Rabbit::nextLife(Cell* ptrNeighborCell[3][3])
     9 {
    10   int numOf[NUM_OF_POSSIBLE_LIFE_TYPES];
    11
    12   numOf[NO_LIFE] = numOf[GRASS] = numOf[RABBIT] = numOf[FOX] = 0;
    13   for (int i=0; i<3; ++i) {
    14     for (int j=0; j<3; ++j) {
    15       ++numOf[ptrNeighborCell[i][j]->life->getType()];
    16     }
    17   }
    18
    19   if (numOf[FOX] > numOf[RABBIT])      // eaten by fox
```

```
    20     return (new NoLife());
    21   else if (age > RABBIT_LIFE_LENGTH)   // natural death
    22     return (new NoLife());
    23   else
    24     return (new Rabbit(age+1));
    25 }
[motoki@x205a]$ cat -n Grass.h
     1 /* Predator-prey-simulationにおいて、                                    */
     2 /* 1つの場所(cell)内に存在する「草(仮想植物)」のクラス Fox (仕様部) */
     3
     4 #ifndef ___Class_Grass
     5 #define ___Class_Grass
     6
     7 #include "LivingThing.h"
     8 #include "Cell.h"
     9
    10 class Grass : public LivingThing {
    11 public:
    12   Grass(): LivingThing(GRASS) {}
    13   LivingThing* nextLife(Cell* ptrNeighborCell[3][3]);
    14 };
    15
    16 #endif
[motoki@x205a]$ cat -n Grass.cpp
     1 /* Predator-prey-simulationにおいて、                                    */
     2 /* 1つの場所(cell)内に存在する「草(仮想植物)」のクラス Fox (実装部) */
     3
     4 #include "Grass.h"
     5 #include "NoLife.h"
     6
     7 // Grass型オブジェクトを操作するための関数群 --------------------------
     8 LivingThing* Grass::nextLife(Cell* ptrNeighborCell[3][3])
     9 {
    10   int numOf[NUM_OF_POSSIBLE_LIFE_TYPES];
    11
    12   numOf[NO_LIFE] = numOf[GRASS] = numOf[RABBIT] = numOf[FOX] = 0;
    13   for (int i=0; i<3; ++i) {
    14     for (int j=0; j<3; ++j) {
    15       ++numOf[ptrNeighborCell[i][j]->life->getType()];
    16     }
    17   }
    18
    19   if (numOf[RABBIT] >= numOf[GRASS])  // eaten by rabbit
    20     return (new NoLife());
    21   else
    22     return (new Grass());
    23 }
[motoki@x205a]$ cat -n NoLife.h
     1 /* Predator-prey-simulationにおいて、1つの場所                          */
     2 /* (cell)内に存在する「生命なし(仮想生物)」のクラス NoLife (仕様部) */
     3
     4 #ifndef ___Class_NoLife
```

```
     5 #define ___Class_NoLife
     6
     7 #include "LivingThing.h"
     8 #include "Cell.h"
     9
    10 class NoLife : public LivingThing {
    11 public:
    12   NoLife(): LivingThing(NO_LIFE) {}
    13   LivingThing* nextLife(Cell* ptrNeighborCell[3][3]);
    14 };
    15
    16 #endif
[motoki@x205a]$ cat -n NoLife.cpp
     1 /* Predator-prey-simulationにおいて、1つの場所                          */
     2 /* (cell)内に存在する「生命なし(仮想生物)」のクラス NoLife(実装部) */
     3
     4 #include "NoLife.h"
     5 #include "Fox.h"
     6 #include "Rabbit.h"
     7 #include "Grass.h"
     8
     9 // NoLife型オブジェクトを操作するための関数群 --------------------------
    10 LivingThing* NoLife::nextLife(Cell* ptrNeighborCell[3][3])
    11 {
    12   int numOf[NUM_OF_POSSIBLE_LIFE_TYPES];
    13
    14   numOf[NO_LIFE] = numOf[GRASS] = numOf[RABBIT] = numOf[FOX] = 0;
    15   for (int i=0; i<3; ++i) {
    16     for (int j=0; j<3; ++j) {
    17       ++numOf[ptrNeighborCell[i][j]->life->getType()];
    18     }
    19   }
    20
    21   if (numOf[FOX] > 1)
    22     return (new Fox());
    23   else if (numOf[RABBIT] > 1)
    24     return (new Rabbit());
    25   else if (numOf[GRASS] > 0)
    26     return (new Grass());
    27   else
    28     return (new NoLife());
    29 }
[motoki@x205a]$ cat -n PredatorPreyWorld.h
     1 /* Predator-prey-simulationにおいて仮想生物達の生存する空間(トーラス */
     2 /* 状に繋がった2次元グリッド空間)のクラス PredatorPreyWorld(仕様部) */
     3
     4 #ifndef ___Class_PredatorPreyWorld
     5 #define ___Class_PredatorPreyWorld
     6
     7 #include "LivingThing.h"
     8 #include "Cell.h"
     9
```

```
    10 const int WORLD_SIZE = 16;
    11
    12 class PredatorPreyWorld {
    13   Cell place[WORLD_SIZE][WORLD_SIZE];
    14 public:
    15   PredatorPreyWorld();
    16   LivingThing* getNextLifeAtPlace(const int longitude, const int latitude);
    17   void setLifeAtPlace(const int longitude, const int latitude,
                                                          LivingThing* life);
    18   void setLifeInformationEmpty();
    19   void printConfiguration() const;
    20 };
    21
    22 #endif
[motoki@x205a]$ cat -n PredatorPreyWorld.cpp
     1 /* Predator-prey-simulationにおいて仮想生物達の生存する空間 (トーラス */
     2 /* 状に繋がった2次元グリッド空間) のクラス PredatorPreyWorld (実装部) */
     3
     4 #include <iostream>
     5 #include <cstddef>
     6 #include "PredatorPreyWorld.h"
     7 using namespace std;
     8
     9 // コンストラクタ ----------------------------------------------------
    10 PredatorPreyWorld::PredatorPreyWorld()
    11 {
    12   for (int i=0; i<WORLD_SIZE; ++i) {
    13     for (int j=0; j<WORLD_SIZE; ++j) {
    14       place[i][j].life = NULL;
    15       for (int x=WEST; x<=EAST; ++x) {
    16         for (int y=SOUTH; y<=NORTH; ++y) {
    17           place[i][j].ptrNeighborCell[x][y]
    18             = &place[(i+x-1+WORLD_SIZE) % WORLD_SIZE]
    19                     [(j+y-1+WORLD_SIZE) % WORLD_SIZE];
    20         }
    21       }
    22     }
    23   }
    24 }
    25
    26 // PredatorPreyWorld型オブジェクトを操作するための関数群 -------------
    27
    28 //World内の引数で指定された場所に次の時点で生存する仮想生物を割り出す
    29 LivingThing* PredatorPreyWorld::getNextLifeAtPlace(const int longitude,
                                                          const int latitude)
    30 {
    31   return place[longitude][latitude].nextLife();
    32 }
    33
    34 //World内の引数で指定された場所に指定された仮想生物オブジェクトを配置
    35 void PredatorPreyWorld::setLifeAtPlace(const int longitude, const int latitude,
    36                                        LivingThing* life)
```

```
    37 {
    38   place[longitude][latitude].life = life;
    39 }
    40
    41 //World内の仮想生物オブジェクトを全て消去する
    42 void PredatorPreyWorld::setLifeInformationEmpty()
    43 {
    44   for (int i=0; i<WORLD_SIZE; ++i) {
    45     for (int j=0; j<WORLD_SIZE; ++j) {
    46       delete place[i][j].life;
    47       place[i][j].life = NULL;
    48     }
    49   }
    50 }
    51
    52 //Worldの状態を出力
    53 void PredatorPreyWorld::printConfiguration() const
    54 {
    55   char printName[NUM_OF_POSSIBLE_LIFE_TYPES] = { '_', ':', 'r', 'F' };
    56
    57   for (int i=WORLD_SIZE-1; i>=0; --i) {
    58     for (int j=0; j<WORLD_SIZE; ++j) {
    59       if (place[i][j].life != NULL)
    60         cout << printName[place[i][j].life->getType()];
    61       else
    62         cout << '?';
    63     }
    64     cout << endl;
    65   }
    66 }
[motoki@x205a]$
```

これに関して、

- LivingThing.h 内で Cell.h のインクルードの指示をしても、

  $$\texttt{LivingThing.h} \xrightarrow{\text{include}} \texttt{Cell.h} \xrightarrow{\text{include(回避)}} \texttt{LivingThing.h}$$

  という風にインクルードガードによって include の無限連鎖は回避され、クラス LivingThing が未定義の状態でクラス Cell の定義を行おうとしてエラーになるだけである。そこで、LivingThing.h では、8 行目 に

  ```
  class Cell;
  ```

  という**前方宣言** (forward declaration) を置いている。クラスに属するメンバの情報をコンパイラに明示する必要がない場合は、この場合の様に include の必要はなく、前方宣言で十分である。

- PredatorPreyWorld.cpp の 55 行目 で指定されている様に、16×16 のマス目全体の生物生息の分布を出力する際は、空 (i.e. 生息なし), 草, 兎, 狐 の状態をそれぞれ "_", ":", "r", "F" という文字で表すことにしている。

(7 つのクラスを利用してシミュレーションを行うコード) プログラミング・デバッグの際に利用した Makefile、main() 関数を含み状態遷移のシミュレーションを行うコード simulateBirthDeathProcess.cpp、16×16 のマス目の初期状態を設定する関数を含むコード setInitialWorld1.cpp、およびコンパイル・実行の様子を次に示す。

```
[motoki@x205a]$ cat -n Makefile
     1 # Makefile for Predator-Prey simulation:
     2
     3 CC    = g++
     4
     5 OBJS1  = simulateBirthDeathProcess.o setInitialWorld1.o \
     6          Fox.o  Rabbit.o  Grass.o  NoLife.o \
     7          PredatorPreyWorld.o
     8
     9 OBJS2  = simulateBirthDeathProcess.o setInitialWorld2.o \
    10          Fox.o  Rabbit.o  Grass.o  NoLife.o \
    11          PredatorPreyWorld.o
    12
    13 OBJS3  = simulateBirthDeathProcess.o setInitialWorld3.o \
    14          Fox.o  Rabbit.o  Grass.o  NoLife.o \
    15          PredatorPreyWorld.o
    16
    17 HEADERS = LivingThing.h Fox.h Rabbit.h Grass.h NoLife.h \
    18           Cell.h PredatorPreyWorld.h
    19
    20 exec_sim1: ${OBJS1}
    21 [ tab ] ${CC} -o exec_sim1 ${OBJS1}
    22 exec_sim2: ${OBJS2}
    23 [ tab ] ${CC} -o exec_sim2 ${OBJS2}
    24 exec_sim3: ${OBJS3}
    25 [ tab ] ${CC} -o exec_sim3 ${OBJS3}
    26
    27 all: exec_sim1 exec_sim2 exec_sim3
    28
    29 simulateBirthDeathProcess.o : simulateBirthDeathProcess.cpp ${HEADERS}
    30 [ tab ] ${CC} -c simulateBirthDeathProcess.cpp
    31 setInitialWorld1.o : setInitialWorld1.cpp ${HEADERS}
    32 [ tab ] ${CC} -c setInitialWorld1.cpp
    33 setInitialWorld2.o : setInitialWorld2.cpp ${HEADERS}
    34 [ tab ] ${CC} -c setInitialWorld2.cpp
    35 setInitialWorld3.o : setInitialWorld3.cpp ${HEADERS}
    36 [ tab ] ${CC} -c setInitialWorld3.cpp
    37 Fox.o : Fox.cpp Fox.h LivingThing.h Cell.h NoLife.h
    38 [ tab ] ${CC} -c Fox.cpp
    39 Rabbit.o : Rabbit.cpp Rabbit.h LivingThing.h Cell.h NoLife.h
    40 [ tab ] ${CC} -c Rabbit.cpp
    41 Grass.o : Grass.cpp Grass.h LivingThing.h Cell.h NoLife.h
    42 [ tab ] ${CC} -c Grass.cpp
    43 NoLife.o : NoLife.cpp NoLife.h Fox.h Rabbit.h Grass.h LivingThing.h Cell.h
    44 [ tab ] ${CC} -c NoLife.cpp
    45 PredatorPreyWorld.o: PredatorPreyWorld.cpp PredatorPreyWorld.h \
                                                          LivingThing.h Cell.h
    46 [ tab ] ${CC} -c PredatorPreyWorld.cpp
    47
    48 clean:
    49 [ tab ] for i in *.o exec_sim1 exec_sim2 exec_sim3 ; do \
    50 [ tab ]   if [ -f $$i ] ; then rm $$i ; fi \
```

```
    51 [ tab ] done
[motoki@x205a]$ cat -n simulateBirthDeathProcess.cpp
     1 /* トーラス状に繋がった２次元グリッド空間において、              */
     2 /* 仮想生物達 (狐, 兎, 草) の出生・死滅の過程をシミュレートする。 */
     3
     4 #include <iostream>
     5 #include "Cell.h"
     6 #include "LivingThing.h"
     7 #include "Fox.h"
     8 #include "Rabbit.h"
     9 #include "Grass.h"
    10 #include "NoLife.h"
    11 #include "PredatorPreyWorld.h"
    12 using namespace std;
    13
    14 int setInitialWorld(PredatorPreyWorld* ptrWorld);
    15
    16 const int MAX_TIME = 5;
    17
    18 int main()
    19 {
    20   PredatorPreyWorld world1, world2;
    21   PredatorPreyWorld* ptrCurrentWorld=&world1, *ptrNextWorld=&world2, *ptrTmp;
    22   LivingThing* ptrLife;
    23
    24   // initialize current world
    25   setInitialWorld(ptrCurrentWorld);
    26
    27   //出生・死滅の過程をシミュレート
    28   for (int time=0; time<=MAX_TIME; ++time) {
    29     //現在の World の状態を出力
    30     cout << "---" << endl
    31          << "time=" << time << endl;
    32     ptrCurrentWorld->printConfiguration();
    33     //次の時点の World の状態を割り出す
    34     for (int i=0; i<WORLD_SIZE; ++i) {
    35       for (int j=0; j<WORLD_SIZE; ++j) {
    36         ptrLife = ptrCurrentWorld->getNextLifeAtPlace(i, j);
    37         ptrNextWorld->setLifeAtPlace(i, j, ptrLife);
    38       }
    39     }
    40     //時間を進める
    41     ptrTmp          = ptrCurrentWorld;
    42     ptrCurrentWorld = ptrNextWorld;
    43     ptrNextWorld    = ptrTmp;
    44     ptrNextWorld->setLifeInformationEmpty();
    45   }
    46 }
[motoki@x205a]$ cat -n setInitialWorld1.cpp
     1 /* トーラス状に繋がった２次元グリッド空間において、                */
     2 /* 仮想生物達 (狐, 兎, 草) の出生・死滅の過程をシミュレートする際の、 */
     3 /* 空間内の仮想生物達の初期配置を設定する関数を保有するモジュール */
```

```
     4
     5 #include "LivingThing.h"
     6 #include "Fox.h"
     7 #include "Rabbit.h"
     8 #include "Grass.h"
     9 #include "NoLife.h"
    10 #include "PredatorPreyWorld.h"
    11
    12 int setInitialWorld(PredatorPreyWorld* ptrWorld)
    13 {
    14   for (int i=0; i<WORLD_SIZE; ++i) {
    15     for (int j=0; j<WORLD_SIZE; ++j) {
    16       switch ((i/4+j/4) % 4) {
    17       case 0:
    18         ptrWorld->setLifeAtPlace(i, j, new Fox());
    19         break;
    20       case 1:
    21         ptrWorld->setLifeAtPlace(i, j, new Grass());
    22         break;
    23       case 2:
    24         ptrWorld->setLifeAtPlace(i, j, new Rabbit());
    25         break;
    26       case 3:
    27         ptrWorld->setLifeAtPlace(i, j, new NoLife());
    28         break;
    29       }
    30     }
    31   }
    32 }
[motoki@x205a]$ make exec_sim1
g++ -c simulateBirthDeathProcess.cpp
g++ -c setInitialWorld1.cpp
g++ -c Fox.cpp
g++ -c Rabbit.cpp
g++ -c Grass.cpp
g++ -c NoLife.cpp
g++ -c PredatorPreyWorld.cpp
g++ -o exec_sim1 simulateBirthDeathProcess.o setInitialWorld1.o Fox.o
                                Rabbit.o  Grass.o  NoLife.o PredatorPreyWorld.o
[motoki@x205a]$ ./exec_sim1
---
time=0
____FFFF::::rrrr
____FFFF::::rrrr
____FFFF::::rrrr
____FFFF::::rrrr
rrrr____FFFF::::
rrrr____FFFF::::
rrrr____FFFF::::
rrrr____FFFF::::
::::rrrr____FFFF
::::rrrr____FFFF
```

```
::::rrrr____FFFF
::::rrrr____FFFF
FFFF::::rrrr____
FFFF::::rrrr____
FFFF::::rrrr____
FFFF::::rrrr____
---
time=1
FFFFF__F:::_rrrr
r__F____::::rrrr
r__F____::::rrrr
rrrFF__F::::rrrr
rrrrFFFFF__F:::_
rrrrr__F____::::
rrrrr__F____::::
rrrrrrrFF__F::::
:::_rrrrFFFFF__F
::::rrrrr__F____
::::rrrrr__F____
::::rrrrrrrFF__F
F__F:::_rrrrFFFF
____::::rrrrr__F
____::::rrrrr__F
F__F::::rrrrrrrF
---
time=2
FFFFF::F:::rrrrr
rFFFF__:::::rrrr
rrFFF__:::::rrrr
rrrFFFFF::::rrrr
rrrrFFFFF::F:::r
rrrrrFFFF__:::::
rrrrrrFFF__:::::
rrrrrrrFFFFF::::
:::rrrrrFFFFF::F
::::rrrrrFFFF__:
::::rrrrrrFFF__:
::::rrrrrrrFFFFF
F::F:::rrrrrFFFF
F__:::::rrrrrFFF
F__:::::rrrrrrFF
FFFF::::rrrrrrrF
---
time=3
____F::F:::rrrrr
r____F::::::rrrr
rr___FF:::::rrrr
rrr____F::::rrrr
rrrr____F::F:::r
rrrrr____F::::::
rrrrrr___FF:::::
rrrrrrr____F::::
```

```
:::rrrrr____F::F
::::rrrrr____F::
::::rrrrrr___FF:
::::rrrrrrr____F
F::F:::rrrrr____
_F::::::rrrrr___
_FF:::::rrrrrr__
___F::::rrrrrrr_
---
time=4
r__FF::F:::rrrrr
rr__FF::::::rrrr
rrr_FFF:::::rrrr
rrrr_FFF::::rrrr
rrrrr__FF::F:::r
rrrrrr__FF::::::
rrrrrrr_FFF:::::
rrrrrrrr_FFF::::
:::rrrrrr__FF::F
::::rrrrrr__FF::
::::rrrrrrr_FFF:
::::rrrrrrrr_FFF
F::F:::rrrrrr__F
FF::::::rrrrrr__
FFF:::::rrrrrrr_
_FFF::::rrrrrrrr
---
time=5
rFFFF::F:::r____
rrrF__::::::____
rrrFF__:::::____
rrrrFF_F::::____
____rFFFF::F:::r
____rrrF__::::::
____rrrFF__:::::
____rrrrFF_F::::
:::r____rFFFF::F
::::____rrrF__::
::::____rrrFF__:
::::____rrrrFF_F
F::F:::r____rFFF
__::::::____rrrF
F__:::::____rrrF
FF_F::::____rrrr
[motoki@x205a]$
```

これに関して、

- Makefile の 27 行目 は、3 つの実行ファイル exec_sim1, exec_sim2, exec_sim3 を全て生成するための規則を表している。

演習問題

□演習 5.1 (既定義クラスの拡張, コンストラクタ, 継承, 多重定義, 暗黙の実引数) 次のC++プログラムを実行するとどういう出力が得られるか? 下の [　　] の部分に予想される出力文字列を入れよ。 但し、ここでは空白は ␣ と明示せよ。

```
[motoki@x205a]$ cat A.h
#ifndef ___Class_A
#define ___Class_A

class A {
protected:
  int a;
public:
  A(int a = 1);
  A(double x);
  void func1();
  void func2();
};

#endif
[motoki@x205a]$ cat A.cpp
#include <iostream>
#include "A.h"
using namespace std;

A::A(int a): a(a)
{
  cout << "A(" << a << ") is closed."
       << endl;
}

A::A(double x): a(static_cast<int>(x))
{
  cout << "A(" << x << ") is closed."
       << endl;
}

void A::func1()
{
  cout << "func1() in class A is called. a="
       << a << endl;
}

void A::func2()
{
  cout << "func2() in class A is called. a^2="
       << a*a << endl;
}
[motoki@x205a]$ cat B.h
#ifndef ___Class_B
#define ___Class_B

#include "A.h"

class B : public A {
  int b;
public:
  B(int b);
  B(int a, int b);
  void func1();
};

#endif
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
[motoki@x205a]$ cat B.cpp
#include <iostream>
#include "B.h"
using namespace std;

B::B(int b): A(), b(b)
{
  cout << "   B(" << b << ") is closed."
       << endl;
}

B::B(int a, int b): A(a), b(b)
{
  cout << "   B(" << a << ", " << b
       << ") is closed." << endl;
}

void B::func1()
{
  cout << "func1() in class B is called. a="
       << a << ", b=" << b << endl;
}
[motoki@x205a]$ cat report18_4_1.cpp
#include <iostream>
#include "A.h"
#include "B.h"
using namespace std;

int main()
{
  cout << "(1)"; A obj;

  cout << "(2)"; A objA(3);
  cout << "(3)"; objA.func1();
  cout << "(4)"; objA.func2();

  cout << "(5)"; B objB(10, 55);
  cout << "(6)"; objB.func1();
  cout << "(7)"; objB.func2();
}
[motoki@x205a]$ g++ report18_4_1.cpp
                                A.cpp B.cpp
[motoki@x205a]$ ./a.out
```

```
[motoki@x205a]$
```

□**演習 5.2 (抽象クラス, 多態性)** 次の C++プログラムを実行するとどういう出力が得られるか？ 下の [　　] の部分に予想される出力文字列を入れよ。 但し、ここでは空白は ␣ と明示せよ。

```
[motoki@x205a]$ cat ArithBinOperator.h
#ifndef ___Class_ArithBinOperator
#define ___Class_ArithBinOperator

#include <string>

class ArithBinOperator {
  std::string symbol;
protected:
  ArithBinOperator(std::string symbol)
                          : symbol(symbol) {}
public:
  virtual int calc(int x, int y) = 0;
  std::string toString() { return symbol; }
};

#endif
[motoki@x205a]$ cat Addition.h
#ifndef ___Class_Addition
#define ___Class_Addition

#include "ArithBinOperator.h"

class Addition : public ArithBinOperator {
public:
  Addition(): ArithBinOperator("+") {}
  int calc(int x, int y) { return x+y; }
};

#endif
[motoki@x205a]$ cat Subtraction.h
#ifndef ___Class_Subtraction
#define ___Class_Subtraction

#include "ArithBinOperator.h"

class Subtraction : public ArithBinOperator {
public:
  Subtraction(): ArithBinOperator("-") {}
  int calc(int x, int y) { return x-y; };
};

#endif
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
[motoki@x205a]$ cat report18_4_2.cpp
#include <iostream>
#include "ArithBinOperator.h"
#include "Addition.h"
#include "Subtraction.h"
using namespace std;

void testWith_1_2_3(ArithBinOperator* ptrOp);

int main()
{
  testWith_1_2_3(new Addition);
  testWith_1_2_3(new Subtraction);
}

void testWith_1_2_3(ArithBinOperator* ptrOp)
{
  cout << "演算子" << ptrOp->toString()
       << "が結合律を満たすかどうか："
       << endl;
  cout << "(1" << ptrOp->toString() << "2)"
       << ptrOp->toString() << "3="
       << ptrOp->calc(ptrOp->calc(1, 2), 3)
       << endl;
  cout << "1" << ptrOp->toString()
       << "(2" << ptrOp->toString() << "3)="
       << ptrOp->calc(1, ptrOp->calc(2, 3))
       << endl;
}
[motoki@x205a]$ g++ report18_4_2.cpp
[motoki@x205a]$ ./a.out
```

```
[motoki@x205a]$
```

□演習 5.3 (抽象クラス, 多態性) 次のC++プログラムを実行するとどういう出力が得られるか？下の会話の様子中で [　　] の部分に予想される出力文字列を入れよ。但し、解答の際は空白を ␣ と明示せよ。下の会話の様子では、下線部はキーボードからの入力を表している。

```
[motoki@x205a]$ cat Point2D.h
#ifndef ___Class_Point2D
#define ___Class_Point2D

#include <string>

class Point2D {
  double x;
  double y;
public:
  Point2D(double x, double y): x(x), y(y) {}
  double getX() const { return x; }
  double getY() const { return y; }
  std::string toString() const ;
};

#endif
[motoki@x205a]$ cat Point2D.cpp
#include <sstream>
#include <iomanip>
#include <string>
#include "Point2D.h"
using namespace std;

string Point2D::toString() const
{
  ostringstream os;
  os << fixed << setprecision(1)
     << "(" << x << ", " << y << ")";
  return os.str();
}
[motoki@x205a]$ cat DistanceCalcMethod.h
#ifndef ___Class_DistanceCalcMethod
#define ___Class_DistanceCalcMethod

#include <string>
#include "Point2D.h"

class DistanceCalcMethod {
  std::string distName;
protected:
  DistanceCalcMethod(std::string distName)
                          : distName(distName) {}
  virtual double op1(double a, double b)
                                       const = 0;
  virtual double op2(double a) const = 0;
public:
  std::string getDistName()
                             { return distName; }
  double getDistanceBetween(Point2D p1,
                                    Point2D p2);
};

#endif
[motoki@x205a]$ cat DistanceCalcMethod.cpp
#include "DistanceCalcMethod.h"

double DistanceCalcMethod::
        getDistanceBetween(Point2D p1, Point2D p2)
{
  return op2(op1(p1.getX(), p2.getX())
              + op1(p1.getY(), p2.getY()));
}
[motoki@x205a]$ cat ManhattanDistCalcMethod.h
#ifndef ___Class_ManhattanDistCalcMethod
#define ___Class_ManhattanDistCalcMethod

#include <cmath>
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
class ManhattanDistCalcMethod : public DistanceCalcMethod {
  double op1(double a, double b) const { return std::fabs(a-b); }
  double op2(double a) const { return a; }
public:
  ManhattanDistCalcMethod()
                      : DistanceCalcMethod("Manhattan distance") {}
};

#endif
[motoki@x205a]$ cat EuclideanDistCalcMethod.h
#ifndef ___Class_EuclideanDistCalcMethod
#define ___Class_EuclideanDistCalcMethod

#include <cmath>

class EuclideanDistCalcMethod : public DistanceCalcMethod {
  double op1(double a, double b) const { return (a-b)*(a-b); }
  double op2(double a) const { return std::sqrt(a); }
public:
  EuclideanDistCalcMethod()
                      : DistanceCalcMethod("Euclidean distance") {}
};

#endif
[motoki@x205a]$ cat test1808_5.cpp
#include <iostream>
#include <iomanip>
#include "Point2D.h"
#include "DistanceCalcMethod.h"
#include "ManhattanDistCalcMethod.h"
#include "EuclideanDistCalcMethod.h"
using namespace std;

void showDistanceBetween(Point2D p1, Point2D p2,
                                 DistanceCalcMethod* ptrToMethod);

int main()
{
  Point2D p1(3.0, 5.0), p2(6.0, 9.0), p3(15.0, 10.0);
  DistanceCalcMethod* ptrToDistCalcMethod
                                  = new ManhattanDistCalcMethod();
  showDistanceBetween(p1, p2, ptrToDistCalcMethod);
  showDistanceBetween(p1, p3, ptrToDistCalcMethod);
  delete ptrToDistCalcMethod;
  ptrToDistCalcMethod = new EuclideanDistCalcMethod();
  showDistanceBetween(p1, p2, ptrToDistCalcMethod);
  showDistanceBetween(p1, p3, ptrToDistCalcMethod);
}

void showDistanceBetween(Point2D p1, Point2D p2,
                                   DistanceCalcMethod* ptrToMethod)
{
  cout << ptrToMethod->getDistName() << " between "
       << p1.toString() << " and " << p2.toString()
       << " = " << fixed << setprecision(1)
       << ptrToMethod->getDistanceBetween(p1, p2) << endl;
}
[motoki@x205a]$ g++ test1808_5.cpp Point2D.cpp DistanceCalcMethod.cpp
[motoki@x205a]$ ./a.out
```

```
[motoki@x205a]$
```

□演習 5.4 (sort モジュールの動作テスト, 計算効率比較) 前ターム「プログラミング AI」実習課題 4 を、C++の下でやり直してみよ。すなわち、

① 例題 5.8 で定義した抽象クラス `SortModuleForIntArray` の派生クラスとして quick-sort モジュールの派生クラスと”btree-traversesort”モジュールの派生クラスを定義し、

② これらのインスタンスとしてできる整列化モジュールの動作テストや動作速度計測を例題 5.9 ∼5.10 で示された `TesterForSortModuleIntArray` クラス や `TimerForSortModuleIntArray` クラス を用いて行うための C++プログラム、及び `Makefile` を作成

してみよ。

6 パラメータ付きのクラス定義、総称的プログラミング、STL

- パラメータ付きのクラス定義
- 型パラメータ付きの関数定義
- 標準テンプレートライブラリ (STL)

6.1 パラメータ付きのクラス定義

{Pohl(1999)7.1 節,7.3-7.4 節 }

例えば例 3.7 で定義した「char 型データを要素とするスタック」のクラス StackChar を考える。このクラスに類似したクラスとして「double 型データを要素とするスタック」のクラスを構成したい場合は、単に、既存の StackChar.h, StackChar.cpp というコード中の「char」を機械的に「double」に置き換え、必要な調整を少し加えるだけで良い。楽と言えば楽なのだが、これを繰り返すと本質的に同じコードが多数できるので、クラスを定義する時間の無駄、保持するコードの無駄という一面もある。また、修正が必要になった場合は定義した全てのコードに対して必要な修正を加えることになるので、保守も面倒になる。

こういった無駄を排し、本質的に同じコードを 1 つにまとめ上げるために、C++言語では、扱うデータの型等をパラメータ化したクラス定義ができる様になっている。

例 6.1 (扱うデータの型をパラメータ化したスタック) 例 3.7 で定義した「char 型データを要素とするスタック」のクラス StackChar において、保持する要素データの型の部分をパラメータ化して得られたクラス定義の例を次に示す。(ここで、プログラム中の下線は StackChar.h, StackChar.cpp からの主要な変更箇所を表す。)

```
[motoki@x205a]$ cat -n Stack.h
     1 /* 「扱うデータの型をパラメータ化したスタック」オブジェクトのクラス */
     2 /*  Stack の仕様部                                                  */
     3
     4 #ifndef ___Class_Stack
     5 #define ___Class_Stack
     6
     7 template<typename TYPE>
     8 class Stack {
     9   static int numOfInstances;
                        // これまでに生成した Stack<TYPE> インスタンスの個数
    10   const int id;     // インスタンスに固有の id 番号
    11   TYPE* element;    // スタック領域へのポインタ
    12   int  size;        // スタックの容量
    13   int  top;         // スタックの top 要素を保持する配列要素の番号
    14 public:
    15   explicit Stack(int size = 100); // スタック容量=size として初期化。
    16                                   //(デフォルトコンストラクタの役割も)
    17   Stack(const Stack<TYPE>& stack);       // コピーコンストラクタ。
```

```
18   Stack(const TYPE a[], const int arraySize);
19                              // 引数の配列要素をスタックに入れて初期化。
20   Stack(int size, const TYPE a[], //容量がsizeのスタック領域を確保し
21         const int arraySize);     //そこに引数の配列要素を入れて初期化
22   ~Stack() { delete[] element; }
23   // オブジェクト(もしくはStack<TYPE>クラス全体)の情報
                                                 // を提供するための関数群
24   static int getNumOfInstances() { return numOfInstances; }
25             // これまでに生成したStack<TYPE>インスタンスの個数を返す
26   int getId() const { return id; }          // スタックのid番号を返す
27   int getSize() const { return size; }      // スタック容量を返す。
28   int getNumOfElements() const { return top+1; }
                                                 //スタック内の要素数を返す
29   void showContents() const;                // スタックの状況を出力。
30   // Stack<TYPE>型オブジェクトを操作するための関数群
31   void reset() { top = -1; }                // スタックを空にする。
32   void resize(int size);                    // スタック容量を変更。
33   bool isEmpty() const { return (top < 0); }  //スタックが空か否か。
34   void pushdown(TYPE ele);                  // pushdown操作。
35   TYPE popup();                             // popup操作。
36 };
37
38
39 //=========================================================================
40 /* 「扱うデータの型をパラメータ化したスタック」オブジェクトのクラス */
41 /*  Stack の実装部                                                       */
42
43 #include <iostream>
44 #include <typeinfo>
45 #include <cstdlib>    // for exit()
46 #include <cstring>    // for memcpy()
47 //#include "Stack.h"
48 //using namespace std;
49
50 // static変数の初期化 ----------------------------------------
51 template<typename TYPE>
52 int Stack<TYPE>::numOfInstances = 0;
53
54 // 各種コンストラクタ ----------------------------------------
55 template<typename TYPE>       //スタック容量=sizeとして初期化
56 Stack<TYPE>::Stack(int size)  //(デフォルトコンストラクタの役割も)
57             : id(numOfInstances++), size(size), top(-1)
58 {
```

```
 59   element = new TYPE[size];
 60 }
 61
 62 template<typename TYPE>       //コピーコンストラクタ
 63 Stack<TYPE>::Stack(const <TYPE>& stack)
 64             : id(numOfInstances++), size(stack.size), top(stack.top)
 65 {
 66   element = new TYPE[stack.size];
 67   memcpy(element, stack.element, sizeof(TYPE)*stack.size);
 68 }
 69
 70 template<typename TYPE>       //引数の配列要素をスタックに入れて初期化
 71 Stack<TYPE>::Stack(const TYPE a[], const int arraySize)
 72             : id(numOfInstances++), size(arraySize), top(arraySize-1)
 73 {
 74   element = new TYPE[size];
 75   for (int i=0; i<arraySize; i++)
 76     element[i] = a[i];
 77 }
 78
 79 template<typename TYPE>        //容量が size のスタック領域を確保し、
 80 Stack<TYPE>::Stack(int size,   //そこに引数の配列要素を入れて初期化。
 81                    const TYPE a[], const int arraySize)
 82             : id(numOfInstances++), size(size), top(arraySize-1)
 83 {
 84   element = new TYPE[size];
 85   for (int i=0; i<arraySize; i++)
 86     element[i] = a[i];
 87 }
 88
 89 // オブジェクトの情報を提供するための関数群 ----------------------------
 90 template<typename TYPE>        // スタックの内容を表示
 91 void Stack<TYPE>::showContents() const
 92 {
 93   std::cout << "stack_of_" << typeid(TYPE).name()
 94        << "(id=" << id << ",size=" << size
 95        << ") has " << top+1 << " elements as follows:" << std::endl;
 96   std::cout << "  [Bottom] ";
 97   for (int i=0; i<=top; i++)
 98     std::cout << element[i] << "  ";
 99   std::cout << "<--" << std::endl;
100 }
101
```

```
102 // <TYPE>型オブジェクトを操作するための関数群 ----------------------
103 template<typename TYPE>          //スタック容量を変更
104 void Stack<TYPE>::resize(int size)
105 {
106   if (top >= size) {
107     std::cout << "insufficient stack size" << std::endl;
108     exit(1);
109   }
110   TYPE* temp = new TYPE[size];
111   memcpy(temp, element, sizeof(TYPE)*(top+1));
112   delete[] element;
113   this->size = size;
114   element = temp;
115 }
116
117 template<typename TYPE>          //pushdown 操作
118 void Stack<TYPE>::pushdown(TYPE ele)
119 {
120   if (++top >= size) {
121     this->resize(size+10);
122   }
123   element[top] = ele;
124 }
125
126 template<typename TYPE>          //popup 操作
127 TYPE Stack<TYPE>::popup()
128 {
129   if (top < 0) {
130     std::cout << "popup from empty stack" << std::endl;
131     exit(1);
132   }
133   return element[top--];
134 }
135
136 #endif
[motoki@x205a]$
```

これに関して、

- 上のクラス定義が為されている時、「char 型データを要素とするスタック」のクラスは Stack<char> 、「double 型データを要素とするスタック」のクラスは Stack<double> 、... と表すことができる。
- 一般に、パラメータ付きのクラス定義を行う場合は、上で示した様に、ヘッダファイルの中に仕様部だけでなく実装部も含める方が無難である。

その理由：ソースファイルをこれまでのやり方に従って仕様部 Stack.h と実装部 Stack.cpp に分けて分割コンパイルを行なっても、「g++ -c Stack.cpp」によって出来る Stack.o はパラメータ付きクラスのメンバ関数群に過ぎず、実際に利用したい Stack<char> 等のメンバ関数群ではない。そのため、リンク時に「'Stack<char>::Stack(int)' に対する定義されていない参照です」といった類のエラーとなる。

- プログラム 7 行目,51 行目,55 行目,62 行目,70 行目,79 行目,90 行目,102 行目,116 行目,125 行目 の template<typename TYPE> は、それぞれ、次に続くクラス定義や関数定義等が TYPE という型パラメータをもつことを明示している。これらの記述は単独の文ではなく、次に続く定義を修飾したものなので、次の行の先頭に移して表記されることもあるが、ここでは見易さのためにこれだけで 1 行としている。
- 上記コードは、大雑把な言い方をすると、例 3.7 で示した StackChar.h, StackChar.cpp というコード中の「スタック要素の型を表す char」の部分を機械的に「TYPE」に置き換えてパラメータ化しただけのもの、とほぼ等しい。「char→TYPE」という単純な置き換え以外の主要な加筆箇所を次に列挙する。
  - ◇ 実装部も (仕様部とともに) ヘッダファイルの中に入れている。(既に上で説明) これに伴って、実装部の最初に置いてあった#include "Stack.h" はコメントアウトしている。(上記コードの 47 行目)
  - ◇ パラメータ化したそれぞれの定義や宣言の前に「template<typename TYPE>」という記述を追加。(既に上で説明)
  - ◇ クラス名を指定する箇所 (class 宣言部、コンストラクタ名以外) では、StackChar クラスのコード中の StackChar をパラメータ化し Stack<TYPE> に置き換えている。
  - ◇ 実装部をヘッダファイルの中に入れたが、ヘッダファイルというものは色々な所からインクルードされる可能性があるので、その中に「using namespace std;」という宣言を含むのは好ましくない。そこで、上記コードの 48 行目 の様に、この宣言はコメントアウトしている。これに伴い、上記コード 93~130 行目 中に現れる「cout」は「std::cout」に置き換え、また「endl」は「std::endl」に置き換えている。
  - ◇ StackChar.h の 19~21 行目 に書かれている StackChar クラスの 2 つのコンストラクタ

    ```
    StackChar(const std::string& str);
    StackChar(int size, const std::string& str);
    ```

    の引数の内 string については、basic_string<TYPE> とパラメータ化できるのかもしれないが、ここでは、string 型変数に相当する char 型配列 (と配列サイズ) が引数として与えられていると見做してパラメータ化を行った。その結果が 上記コードの 18~21 行目 の

    ```
    Stack(const TYPE a[], const int arraySize);
    Stack(int size, const TYPE a[], const int arraySize);
    ```

    である。これに伴って、これらのコンストラクタを実装したコードも多少手直しされている。(上記コードの 70~87 行目)
  - ◇ StackChar.cpp の 49 行目 に書かれている「cout << "stack_of_char"···」という記述中の「char」の部分も TYPE という型パラメータを用いた表現に置き換え

たい。それを行ったのが、上記コード 93 行目 の

```
std::cout << "stack_of_" << typeid(TYPE).name()···
```

という行である。この記述の中の typeid() は、引数で与えられた型名や式の型情報を表す type_info 型オブジェクトへの参照を値とする演算子である。type_info クラスは name() というメンバ関数を持っており、typeid(TYPE).name() は実行時に TYPE に割当てられたデータ型を表す文字列 (処理系に依存) になる。具体的に、g++では、TYPE が int の時は typeid(TYPE).name() の部分は「c」となり、TYPE が double の時は typeid(TYPE).name() の部分は「d」となる。また、これに関連して、typeid() 演算子の部分を正しくコンパイルするためのヘッダファイルを 上記コード 44 行目 でインクルードしている。

◇ スタック内の個々の要素の大きさは例 3.7 の StackChar クラスの場合は 1byte であったが、Stack<TYPE>クラスの場合は sizeof(TYPE)byte となる。それゆえ、上記コード 67 行目,111 行目 では、スタックを構成する配列領域の大きさを「sizeof(TYPE)*(配列の要素数) byte」と見積もって処理を進めている。

- 型パラメータ付きのクラス Stack が上の様に定義されていれば、それを使って次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n useStackDouble.cpp
     1 /* 総称的に定義されたクラス Stack.h の利用例                          */
     2 /* (1) 配列内の double 型データ列をスタックを用いて反転した後出力 */
     3 /* (2) デフォルトコンストラクタの利用                               */
     4 /* (3) コピーコンストラクタの利用                                   */
     5
     6 #include <iostream>
     7 #include "Stack.h"
     8 using namespace std;
     9
    10 int main(void)
    11 {
    12   Stack<double> stackA(100);
    13   double a[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
    14
    15   for (int i=0; i < 5; ++i)
    16     stackA.pushdown(a[i]);
    17   for (int i=0; !stackA.isEmpty(); ++i)
    18     a[i] = stackA.popup();
    19   cout << "Reversed data sequence a = ( ";
    20   for (int i=0; i<5; ++i)
    21     cout << a[i] << "  ";
    22   cout << ")" << endl << "---" << endl;
    23
    24   Stack<double> stackB;       //デフォルトコンストラクタの利用
    25   Stack<double> stack[3];     //デフォルトコンストラクタの利用
    26   for (int i=0; i<3; ++i) {
```

```
    27     for (int k=0; k<=i; ++k)
    28       stack[i].pushdown(99.99);
    29   }
    30   stackB.showContents();
    31   for (int i=0; i<3; ++i)
    32     stack[i].showContents();
    33   cout << "---" << endl;
    34
    35   Stack<double> stackC(a, sizeof(a)/sizeof(double));
    36   Stack<double> stackD(stackC);  //コピーコンストラクタの利用
    37   stackD.pushdown(1.0/0.0);
    38   Stack<double> stackE = stackD; //コピーコンストラクタの利用？
    39   stackE.popup();
    40   stackE.pushdown(0.0/0.0);
    41   stackC.showContents();
    42   stackD.showContents();
    43   stackE.showContents();
    44   cout << "---" << endl;
    45
    46   cout << "生成された Stack<double>インスタンス数 = "
    47        << Stack<double>::getNumOfInstances() << endl;
    48 }
[motoki@x205a]$ g++ useStackDouble.cpp
[motoki@x205a]$ ./a.out
Reversed data sequence a = ( 5.5  4.4  3.3  2.2  1.1  )
---
stack_of_d(id=1,size=100) has 0 elements as follows:
  [Bottom] <--
stack_of_d(id=2,size=100) has 1 elements as follows:
  [Bottom] 99.99  <--
stack_of_d(id=3,size=100) has 2 elements as follows:
  [Bottom] 99.99  99.99  <--
stack_of_d(id=4,size=100) has 3 elements as follows:
  [Bottom] 99.99  99.99  99.99  <--
---
stack_of_d(id=5,size=5) has 5 elements as follows:
  [Bottom] 5.5  4.4  3.3  2.2  1.1  <--
stack_of_d(id=6,size=15) has 6 elements as follows:
  [Bottom] 5.5  4.4  3.3  2.2  1.1  inf  <--
stack_of_d(id=7,size=15) has 6 elements as follows:
  [Bottom] 5.5  4.4  3.3  2.2  1.1  -nan  <--
---
生成された Stack<double>インスタンス数 = 8
```

```
[motoki@x205a]$
```

上で例示した様なパラメータ付きの定義は、利用する際のパラメータ部の**具現化** (instantiation) の指定 (i.e. 値の指定) に従ってコンパイル時に色々な定義を創り出すので、多態性をもたらすとも考えられる。パラメータ付きの定義を用いたプログラミングを一般に**総称的プログラミング** (generic programming, **ジェネリックプログラミング**) と呼ぶ。

(補足, どうしても仕様部と実装部を分けたい場合) 例えば上の例 6.1 で仕様部と実装部を分けて Stack<char>, Stack<double>という具現化を行う場合は、実装部のコード Stack.cpp の最後に次の 19 行を追加しておけば、Stack.cpp のコンパイル結果である Stack.o の中には具現化前のコードだけでなく、Stack<char>のメンバ関数の実行コードも Stack<double>のメンバ関数の実行コードも含まれる様になる。

```
template int Stack<char>::numOfInstances;
template Stack<char>::Stack(int size);
template Stack<char>::Stack(const Stack<char>& stack);
template Stack<char>::Stack(const char a[], const int arraySize);
template Stack<char>::Stack(int size, const char a[], const int arraySize);
template void Stack<char>::showContents() const;
template void Stack<char>::resize(int size);
template void Stack<char>::pushdown(char ele);
template char Stack<char>::popup();

template int Stack<double>::numOfInstances;
template Stack<double>::Stack(int size);
template Stack<double>::Stack(const Stack<double>& stack);
template Stack<double>::Stack(const double a[], const int arraySize);
template Stack<double>::Stack(int size, const double a[], const int arraySize);
template void Stack<double>::showContents() const;
template void Stack<double>::resize(int size);
template void Stack<double>::pushdown(double ele);
template double Stack<double>::popup();
```

パラメータ付きの定義の構文：

- 一般に、定義の先頭位置に

  template < パラメータ指定, ..., パラメータ指定 >

  という記述を挿入することにより、定義内の「データ型」や「整定数」をパラメータ化できる。ここで、パラメータ指定 としては次の形のものが許される。
  - ◇ typename パラメータ名
  - ◇ class パラメータ名
  - ◇ int パラメータ名
  - ◇ typename パラメータ名 = デフォルトのデータ型名
  - ◇ class パラメータ名 = デフォルトのデータ型名
  - ◇ int パラメータ名 = デフォルトの整定数値

  この内、キーワード typename, class に続くパラメータ名はどちらも「データ型」名を表すパラメータとして定義内で用いることができ、キーワード int に続くパラメータ名は「整定数」値を表すパラメータとして定義内で用いることができる。

パラメータ付きのクラス定義に関する諸注意：

- パラメータ付きのクラス (テンプレートクラスもしくはクラステンプレートという) を利用したい場合、具現化したクラスを

  クラス名 < パラメータ指定 , . . . , パラメータ指定 >

  という形で表す。
- 一般に、パラメータ付きのクラス定義を行う場合は、ヘッダファイルの中に仕様部だけでなく実装部も含める方が無難である。
- テンプレートクラスにフレンド関数を指定することもできる。

  その際、
  ◇ パラメータに依存しないフレンド関数を指定した場合は、そのフレンド関数は全ての具現化クラスのフレンド関数になる。
  ◇ パラメータに依存したフレンド関数を指定した場合は、そのフレンド関数は具現化クラスのフレンド関数になる。
- テンプレートクラス内で宣言された static なデータメンバに関しては、個別の具現化クラス毎にデータ領域が確保される。

  例えば、例 6.1 で定義された Stack クラスの具現化クラスとして Stack<char> と Stack<double> が利用される場合、
  ◇ Stack<char>の static なデータメンバである Stack<char>::numOfInstances と
  ◇ Stack<double>の static なデータメンバである Stack<double>::numOfInstances
  は別の変数領域として確保され、具現化クラス毎に通しの id 番号が振り分けられることになる。
- 整定数値を表すパラメータは、テンプレートクラス内の配列メンバの要素数をパラメータ化したい場合等に利用できる。

例 6.2 (添字の有効性をチェックする機能を備えた配列) 指定されたパラメータ TYPE に対して「添字の有効性をチェックする機能を備えた TYPE 型配列」のクラスをもたらすテンプレートクラスの定義例を次に示す。

```
[motoki@x205a]$ cat -n SafeArray.h
     1 /* 「添字の有効性をチェックする機能を備えた配列」                    */
     2 /* をインスタンスとする型パラメータ付きクラス SafeArray の仕様部   */
     3
     4 #ifndef ___Class_SafeArray
     5 #define ___Class_SafeArray
     6
     7 template<typename TYPE>
     8 class SafeArray {
     9   TYPE* basePtr;                  //pointer to the 1st element
    10   int size;                       //array size
    11 public:
    12   explicit SafeArray(int n=100);        //create a SafeArray of size n
    13   SafeArray(const SafeArray<TYPE>& a);  //copy constructor
    14   SafeArray(const TYPE a[], int n);     //copy a standard array
```

```
15   ~SafeArray() { delete[] basePtr; }
16   int getSize() { return size; }
17   typedef TYPE* iterator;
18   iterator begin() { return basePtr; }
19   iterator end() { return basePtr + size; }
20   TYPE& operator[](int i);              //range-checked element
21   SafeArray<TYPE>& operator=(const SafeArray<TYPE>& a);
22 };
23
24 //========================================================================
25 /* 「添字の有効性をチェックする機能を備えた配列」                          */
26 /* をインスタンスとする型パラメータ付きクラス SafeArray の実装部   */
27
28 #include <cassert>
29
30 template<typename TYPE>        //create a SafeArray of size n
31 SafeArray<TYPE>::SafeArray(int n): size(n)
32 {
33   assert(n > 0);
34   basePtr = new TYPE[size];
35   assert(basePtr != 0);
36 }
37
38 template<typename TYPE>        //copy constructor
39 SafeArray<TYPE>::SafeArray(const SafeArray<TYPE>& a)
40 {
41   size = a.size;
42   basePtr = new TYPE[size];
43   assert(basePtr != 0);
44   for (int i=0; i<size; ++i)
45     basePtr[i] = a.basePtr[i];
46 }
47
48 template<typename TYPE>        //copy a standard array
49 SafeArray<TYPE>::SafeArray(const TYPE a[], int n)
50 {
51   assert(n > 0);
52   size = n;
53   basePtr = new TYPE[size];
54   assert(basePtr != 0);
55   for (int i=0; i<size; ++i)
56     basePtr[i] = a[i];
57 }
```

```
   58
   59 template<typename TYPE>        //range-checked element
   60 TYPE& SafeArray<TYPE>::operator[](int i)
   61 {
   62   assert (0<=i && i<size);
   63   return basePtr[i];
   64 }
   65
   66 template<typename TYPE>        //assignment operator
   67 SafeArray<TYPE>& SafeArray<TYPE>::operator=(const SafeArray<TYPE>& a)
   68 {
   69   assert (a.size == size);
   70   for (int i=0; i<size; ++i)
   71     basePtr[i] = a.basePtr[i];
   72   return *this;
   73 }
   74
   75 #endif
[motoki@x205a]$
```

これに関して、

- プログラム 17~19 行目 で定義されている iterator, begin(), end() に相当するものが、2.15 節で紹介した動的配列のテンプレートクラス vector を始めとした (標準ライブラリ内の) 多くのクラスに備わっている。これらの仕掛けを用いることにより、データ群を保持するための入れ物の実装方法に依存しない形で、データ群の繰り返し処理を記述することが可能となる。
- プログラム 21 行目,66~73 行目 では、全ての配列要素をコピーするための代入演算子 = を定義している。
- 型パラメータ付きのクラス SafeArray が上の様に定義されていれば、それを使って次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n useSafeArrayInt.cpp
     1 // 型パラメータ付きのクラス SafeArray を利用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <string>
     6 #include "SafeArray.h"
     7 using namespace std;
     8
     9 void showContentsOf(string arrayName, SafeArray<int> a);
    10
    11 int main()
    12 {
    13   SafeArray<int> a(10), b(10);
```

```
14
15   for (int i=0; i<a.getSize(); i++) {
16     a[i] = i+10;
17     b[i] = i*i;
18   }
19   showContentsOf("a", a);
20   showContentsOf("b", b);
21
22   a = b;
23   cout << "a = b;" << endl;
24   cout << "==> ";
25   showContentsOf("a", a);
26 }
27
28 void showContentsOf(string arrayName, SafeArray<int> a)
29 {
30   cout << arrayName << ".size=" << a.getSize() << ",  "
31        << arrayName << "=(";
32   for (SafeArray<int>::iterator p=a.begin(); p!=a.end(); ++p)
33     cout << right << setw(4) << *p;
34   cout << ")" << endl;
35 }
[motoki@x205a]$ g++ useSafeArrayInt.cpp
[motoki@x205a]$ ./a.out
a.size=10,  a=(  10  11  12  13  14  15  16  17  18  19)
b.size=10,  b=(   0   1   4   9  16  25  36  49  64  81)
a = b;
==> a.size=10,  a=(   0   1   4   9  16  25  36  49  64  81)
[motoki@x205a]$
```

ここで、

◇ 利用例のプログラム 32~33 行目 では、全ての配列要素を出力する繰り返し処理がデータ集合 {a[0], a[1], ..., a[a.size-1]} を保持する方法に依存しない形で記述されている。この 32~33 行目に現れる変数 p の様に、処理する要素を次々に指す iterator 型の変数を**反復子** (iterator, **イテレータ**) と呼ぶ。

6.2 型パラメータ付きの関数定義

{Pohl(1999)7.2 節 }

通常の関数内に現れるデータ型の部分をパラメータ化することもできる。型パラメータ付きの関数を**テンプレート関数**もしくは**関数テンプレート**と呼ぶ。

例 6.3 (配列全体をコピーする型パラメータ付き関数) 指定された型パラメータ TYPE1, TYPE2 に対して「TYPE2 型配列を TYPE1 型配列に要素単位でコピーするテンプレート関数

copy()」、および指定された型パラメータ TYPE に対して「TYPE 型配列の要素列を出力するテンプレート関数 print()」を定義し使用した例を次に示す。

```
[motoki@x205a]$ cat -n useTemplateCopy.cpp
     1 // 型パラメータ付きの関数 copy() の定義・利用例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <string>
     6 using namespace std;
     7
     8 template<typename TYPE1, typename TYPE2>
     9 void copy(TYPE1 x[], TYPE2 y[], int size);
    10
    11 template<typename TYPE>
    12 void print(string arrayName, TYPE x[], int size);
    13
    14 int main()
    15 {
    16   double a[5] = {1.1, 2.2, 3.3, 4.4, 5.5}, b[5];
    17   int m[5];
    18
    19   print("a", a, 5);
    20   ::copy(b, a, 5);
    21   print("b", b, 5);
    22
    23   ::copy(m, a, 5);
    24   print("m", m, 5);
    25 }
    26
    27 template<typename TYPE1, typename TYPE2>
    28 void copy(TYPE1 x[], TYPE2 y[], int size)
    29 {
    30   for (int i=0; i<size; ++i)
    31     x[i] = y[i];
    32 }
    33
    34 template<typename TYPE>
    35 void print(string arrayName, TYPE x[], int size)
    36 {
    37   cout << arrayName << "[" << size << "] = {" << scientific << fixed;
    38   for (int i=0; i<size-1; ++i)
    39     cout << x[i] << ", ";
    40   cout << x[size-1] << "}" << endl;
```

```
    41 }
[motoki@x205a]$ g++ useTemplateCopy.cpp
[motoki@x205a]$ ./a.out
a[5] = {1.100000, 2.200000, 3.300000, 4.400000, 5.500000}
b[5] = {1.100000, 2.200000, 3.300000, 4.400000, 5.500000}
m[5] = {1, 2, 3, 4, 5}
[motoki@x205a]$
```

ここで、

- プログラム 20 行目,23 行目 でスコープ解決演算子 :: を用いているのは、標準ライブラリ std 内で定義された copy() 関数を適用候補から除外するためである。

例 6.4 (2 つの変数の中身を交換する型パラメータ付き関数) 例 2.5 で定義した関数 void swap(int&, int&) の引数の型の部分をパラメータ化し、指定された型パラメータ TYPE に対して「TYPE 型の 2 つの変数の中身を交換する (テンプレート) 関数 swap()」を定義し使用した例を次に示す。

```
[motoki@x205a]$ cat -n useTemplateSwap.cpp
     1 // 型パラメータ付きの関数 swap() の定義・利用例
     2
     3 #include <iostream>
     4 #include <cstring>
     5 using namespace std;
     6
     7 template<typename TYPE>
     8 void swap(TYPE& x, TYPE& y);
     9
    10 void swap(char* x, char* y);
    11
    12 int main()
    13 {
    14   int i=1, j=22;
    15   double a=33.3, b=444.4;
    16   char str1[]="abcd", str2[]="1234";
    17
    18   ::swap(i, j);
    19   cout << "i=" << i << ", j=" << j << endl;
    20
    21   ::swap(a, b);
    22   cout << "a=" << a << ", b=" << b << endl;
    23
    24   ::swap(str1, str2);
    25   cout << "str1=" << str1 << ", str2=" << str2 << endl;
    26 }
```

```
    27
    28 template<typename TYPE>
    29 void swap(TYPE& x, TYPE& y)
    30 {
    31   TYPE temp;
    32
    33   temp = x;
    34   x = y;
    35   y = temp;
    36 }
    37
    38 void swap(char* x, char* y)
    39 {
    40   int maxLength = max(strlen(x), strlen(y));
    41   char* temp = new char[maxLength+1];
    42
    43   strcpy(temp, x);
    44   strcpy(x, y);
    45   strcpy(y, temp);
    46   delete[] temp;
    47 }
[motoki@x205a]$ g++ useTemplateSwap.cpp
[motoki@x205a]$ ./a.out
i=22, j=1
a=444.4, b=33.3
str1=1234, str2=abcd
[motoki@x205a]$
```

ここで、

- プログラム 18 行目,21 行目,24 行目 でスコープ解決演算子 `::` を用いているのは、標準ライブラリ `std` 内で定義された `swap()` 関数を適用候補から除外するためである。
- プログラム 40 行目 に現れる `max()` は、標準ライブラリ `std` 内で定義された `max()` 関数を表す。
- プログラム 24 行目 の関数呼び出し `::swap(str1, str2);` に対しては、28~36 行目で定義されたテンプレート関数も、38~47 行目で定義された非テンプレート関数も適用可能である。こういう場合は、非テンプレート関数の方が優先的に適用される。一般に、適用可能な関数が複数あった時は、次の順に優先的に選ばれる。
  ① 自明の型変換を行うことによりシグネチャが合致する非テンプレート関数
  ② 適切に型パラメータを設定することによりシグネチャが合致するテンプレート関数
  ③ 非テンプレート関数上での通常の選択手順による関数 (⟶2.11 節を参照)

6.3 標準テンプレートライブラリ (STL)

{Pohl(1999)7.5-7.6 節,7.8 節, シルト (2012)14 章, プログラミング I(2017)23.5 節}

C++言語においては、**標準テンプレートライブラリ** (standard template library, **STL**) と呼ばれるライブラリの中に、標準的なデータ構造やアルゴリズムを実装したコードのテンプレートが用意されていて、それらを組み合わせて自由に総称的プログラミングを進められる様になっている。STL の中核は次の 3 種類の要素から成っている。

- **コンテナ** (container) ··· データ (群) を保持するための入れ物オブジェクト
- **反復子** (iterator, **イテレータ**) ··· ポインタに似たオブジェクトで、コンテナ内の全てのデータを扱う繰り返し処理を、採用したコンテナの実装方法に依存しない形で書き表すことを可能にする。(これを使うと、処理するコードを変えずにコンテナの実装方法を変更することも可能になる。)
- **アルゴリズム** ··· 用意されたコンテナの下での、標準的 (で効率の良い) 操作手段を提供する。

例 6.5 (反復子を用いた繰り返し) 例 2.8 で示した `vector<int>` を利用したプログラム `use _vectorType.cpp` に現れる繰り返し処理の中で、「`v[i]`」という要素表現は配列やベクトルに固有の表現である。そこで、反復子を用いてこれらの繰り返し処理を表し、データ集合を保持する方法に依存しないコードに書き換えた例を次に示す。(ここで、プログラム中の下線は例 2.8 の `use_vectorType.cpp` からの主要な変更箇所を表す。)

```
[motoki@x205a]$ cat -n useSTL_vectorInt_iterator.cpp
     1 // STL 内のテンプレートクラス vector<int> を利用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <vector>
     6 using namespace std;
     7
     8 void showContentsOf(vector<int> v);
     9
    10 int main()
    11 {
    12   vector<int> v;
    13
    14   showContentsOf(v);
    15
    16   for (int i=0; i<10; i++)
    17     v.push_back(i);
    18   showContentsOf(v);
    19
    20   for (int i=0; i<5; i++)
    21     v.pop_back();
```

```
    22   showContentsOf(v);
    23
    24   for (vector<int>::iterator p=v.begin(); p!=v.end(); p++)
    25     *p *= *p;
    26   showContentsOf(v);
    27 }
    28
    29 void showContentsOf(vector<int> v)
    30 {
    31   vector<int>::iterator p=v.begin();
    32
    33   cout << "要素数 = " << v.size() << endl;
    34   cout << "内容   = (";
    35   if (v.size() > 0)
    36     cout << right << setw(2) << *(p++);
    37   for ( ; p!=v.end(); p++)
    38     cout << ", "<< right << setw(2) << *p;
    39   cout << " )" << endl;
    40 }
[motoki@x205a]$ g++ useSTL_vectorInt_iterator.cpp
[motoki@x205a]$ ./a.out
要素数 = 0
内容   = ( )
要素数 = 10
内容   = ( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9 )
要素数 = 5
内容   = ( 0,  1,  2,  3,  4 )
要素数 = 5
内容   = ( 0,  1,  4,  9, 16 )
[motoki@x205a]$
```

ここで、

- プログラム 24 行目 の「vector<int>::iterator p」の部分は、vector<int>型オブジェクト内に保持されている要素を順に巡るための反復子として p という変数を用いることを宣言している。これを用いた場合、p は vector<int>型オブジェクト内の要素へのポインタ (に相当するもの) を表し、++p や p++ は p の指す先を次の要素に移動する操作を表す。また、v が vector<int>型オブジェクトの時、v.begin() はそのオブジェクト内の最初の要素へのポインタ (に相当するもの)、v.end() はそのオブジェクト内の最後の要素の次の場所 (オブジェクト外) へのポインタ (に相当するもの) を表す。
- プログラム 24~25 行目 の部分は例 2.8 では次の様に記述されていた。

```
for (int i=0; i<v.size(); i++)
  v[i] *= v[i];
```

ベクトルや配列に固有の表し方である v[i] という表現が除去されたことが確認さ

れる。

例 6.6 (コンテナ list<double>, ライブラリ numeric) 双方向連結リストで double 型データ群を保持するオブジェクトのクラス list<double>と、汎用数値アルゴリズムのライブラリ numeric 内の関数 accumulate() を使用した例を次に示す。

```
[motoki@x205a]$ cat -n useSTL_listDouble.cpp
     1 // STL 内のテンプレートクラス list<double> を利用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <list>
     6 #include <numeric>        // for accumulate()
     7 using namespace std;
     8
     9 void showContentsOf(list<double>& listD);
    10
    11 int main()
    12 {
    13   double a[5] = {3.3, 2.2, 5.5, 1.1, 4.4};
    14   list<double> z;
    15
    16   for (int i=0; i<5; ++i)
    17     z.push_front(a[i]);
    18   showContentsOf(z);
    19
    20   z.sort();
    21   showContentsOf(z);
    22
    23   cout << "sum = " << accumulate(z.begin(), z.end(), 0.0) << endl;
    24 }
    25
    26 void showContentsOf(list<double>& listD)
    27 {
    28   list<double>::iterator p=listD.begin();
    29
    30   cout << "要素数 = " << listD.size() << endl;
    31   cout << "内容   = (";
    32   if (listD.size() > 0)
    33     cout << *(p++);
    34   for ( ; p!=listD.end(); p++)
    35     cout << ", "<< *p;
    36   cout << " )" << endl;
    37 }
```

```
[motoki@x205a]$ g++ useSTL_listDouble.cpp
[motoki@x205a]$ ./a.out
要素数 = 5
内容   = (4.4, 1.1, 5.5, 2.2, 3.3 )
要素数 = 5
内容   = (1.1, 2.2, 3.3, 4.4, 5.5 )
sum = 16.5
[motoki@x205a]$
```

ここで、

- プログラム 17 行目 に現れる push_front() は list<double>型オブジェクトに備わったメンバ関数で、引数で与えられたデータをオブジェクトの保持するリストの先頭要素の直前に挿入する働きをする。
- プログラム 20 行目 に現れる sort() も list<double>型オブジェクトに備わったメンバ関数で、オブジェクトの保持するデータ群が小さい順に並ぶ様にデータ群を並び替える働きをする。
- プログラム 23 行目 に現れる accumulate() は汎用数値アルゴリズムのライブラリ numeric 内に用意された関数で、(第 1 引数で指された要素) ～(第 2 引数で指された要素の手前の要素) と (第 3 引数の値)　の合計値を計算して返す働きをする。

例 6.7 (コンテナ map<string,int,less<string> >) 数学的な意味での「写像 (map, function) のグラフ」(i.e. 何らかの写像 $f{:}K{\to}V$ の対応関係を表す集合 $\{(x, f(x))|x{\in}K\}$) を表すための map と呼ばれる **(連想) コンテナ**を利用して、標準入力から入力した単語の出現回数を保持した例を次に示す。

```
[motoki@x205a]$ cat -n useSTL_mapStringInt.cpp
     1 // STL内のテンプレートクラスmap<string,int,less<string> >を利用した例
     2 //    単語列を入力し、各単語の出現回数を表示する
     3
     4 #include <iostream>
     5 #include <iomanip>
     6 #include <string>
     7 #include <map>
     8 using namespace std;
     9
    10 int main()
    11 {
    12   map<string,int,less<string> > word_freq;
    13   map<string,int,less<string> >::iterator p;
    14   string word;
    15
    16   //標準入力から単語を次々に入力し、順にword_freq上に記録
    17   cin >> word;
    18   while (cin.good()) {
```

```
    19     if (word[word.size()-1]==',' || word[word.size()-1]=='.'
    20         || word[word.size()-1]==';' || word[word.size()-1]==':')
    21       word = word.substr(0,word.size()-1);
    22     if (word.size() == 0)
    23       continue;
    24     p = word_freq.find(word);
    25     if (p != word_freq.end())
    26       ++(p->second);
    27     else
    28       word_freq.insert(make_pair(word, 1));   //word_freq[word]=1;
    29     cin >> word;
    30   }
    31
    32   //word_freq 内に保存されているデータを表示
    33   cout << word_freq.size() << " distinct words appeared: " << endl;
    34   int num = 0;
    35   cout << "{ ";
    36   p = word_freq.begin() ;
    37   if (p!=word_freq.end()) {
    38     cout << setw(15) << p->first << ":" << setw(3) <<p->second;
    39     ++num;
    40     ++p;
    41   }
    42   for ( ; p != word_freq.end(); ++p, ++num) {
    43     if (num >= 3) {
    44       cout << "," << endl
    45            << "  ";
    46       num = 0;
    47     }else {
    48       cout << ", ";
    49     }
    50     cout << setw(15) << p->first << ":" << setw(3) <<p->second;
    51   }
    52   cout << " }" << endl;
    53
    54   //入力エラーが起きてなかったか確認
    55   if (!cin.eof())
    56     cout << "Warning: input error." << endl;
    57 }
[motoki@x205a]$ g++ useSTL_mapStringInt.cpp
[motoki@x205a]$ cat useSTL_mapStringInt.data
C++, invented at Bell labs by Bjarne Stroustrup in the mid-1980s,
is a powerful modern successor language to C.  C++ adds to C the
```

```
concept of class, a mechanism for providing user-defined types,
also called abstract data types.  C++ supports object-oriented
programming by these means and by providing inheritance and
runtime type binding.
[motoki@x205a]$ ./a.out < useSTL_mapStringInt.data
42 distinct words appeared:
{            Bell:  1,          Bjarne:  1,              C:  2,
              C++:  3,      Stroustrup:  1,              a:  2,
         abstract:  1,            adds:  1,           also:  1,
              and:  2,              at:  1,        binding:  1,
               by:  3,          called:  1,          class:  1,
          concept:  1,            data:  1,            for:  1,
               in:  1,     inheritance:  1,       invented:  1,
               is:  1,            labs:  1,       language:  1,
            means:  1,       mechanism:  1,      mid-1980s:  1,
           modern:  1, object-oriented:  1,             of:  1,
         powerful:  1,     programming:  1,      providing:  2,
          runtime:  1,       successor:  1,       supports:  1,
              the:  2,           these:  1,             to:  2,
             type:  1,           types:  2,   user-defined:  1 }
[motoki@x205a]$
```

ここで、

- プログラム 12 行目 は、(string 型データ, int 型データ) という形の 2 項組のデータ集合を第 1 要素の小さい順 (辞書順) に保持する map<string,int,less<string> >型オブジェクトを表すために word_freq という変数を用いることを宣言している。map 型オブジェクトの場合、保持している 2 項組の第 1 要素はキーとしての役割を持ち、指定したキー値をもつ 2 項組の第 2 要素を検索するための関数が提供される。
- プログラム 12~13 行目 の map<string,int,less<string>␣> という表記の最後の方に現れる半角空白は削除できない。削除するとコンパイルエラーとなる。
- プログラム 19~23 行目 では、標準入力から読み取った単語 word の最後にコンマ (,) やピリオド (.), セミコロン (;), コロン (:) が付いていた場合に、それらを除去する処理を行なっている。
- プログラム 24 行目 に現れる find() は map<string,int,less<string> >型オブジェクト word_freq に備わったメンバ関数で、引数で指定されたキー値をオブジェクト内に保持された 2 項組の中から探し、見つかったらその 2 項組へのポインタ (に相当するもの) を、見つからなかったら word_freq.end() を返す。従って、25~28 行目 では、word が既読の単語であった場合はその度数を 1 だけ大きくし、新規の単語であった場合は 28 行目を実行することになる。
- プログラム 28 行目 に現れる make_pair() は与えられた第 1 引数と第 2 引数を組み合わせて 2 項組を構成する<utility> 内のライブラリ関数である。
- プログラム 28 行目 に現れる insert() も map<string,int,less<string> >型オブジェクトに備わったメンバ関数で、引数で指定された 2 項組をオブジェクト内の然る

べき場所に保存する働きをする。

- プログラム 34~52 行目 では、オブジェクト word_freq の中に保存されている 2 項組を全て出力している。1 行に 3 個の 2 項組を出力するために、34 行目 では現在の出力行に既に出力した 2 項組の個数を保持する変数 num を用意している。
- プログラム 38 行目と 50 行目 に現れる p->first, p->second は、それぞれ p の指す 2 項組の 1 番目の要素, 2 番目の要素を表す。

例 6.8 (STL アルゴリズム内の find(),sort()) STL アルゴリズムのライブラリ algorithm 内の find() 関数 と sort() 関数 を使用した例を次に示す。

```
[motoki@x205a]$ cat -n useSTL_algorithm.cpp
     1 // STL アルゴリズム内の find(), sort() を利用した例
     2
     3 #include <iostream>
     4 #include <string>
     5 #include <vector>
     6 #include <algorithm>
     7 using namespace std;
     8
     9 void showContentsOf(vector<string> wordSeq);
    10
    11 int main()
    12 {
    13   vector<string> wordSeq;
    14   vector<string>::iterator where;
    15   string word;
    16
    17   //標準入力から単語を次々に入力し、順に wordSeq に記録
    18   cin >> word;
    19   while (cin.good()) {
    20     wordSeq.push_back(word);
    21     cin >> word;
    22   }
    23   cout << "入力単語列:" << endl;
    24   showContentsOf(wordSeq);
    25   cout << "---" << endl;
    26
    27   //STL アルゴリズム内の find() を利用
    28   where = find(wordSeq.begin(), wordSeq.end(), "C++");
    29   cout << "最初の\"C++\"以降の 10 単語列:" << endl;
    30   for (int i=0 ; where!=wordSeq.end() && i<10; ++where, ++i)
    31     cout << *where << " ";
    32   cout << endl
    33        << "---" << endl;
```

```
    34
    35   //STL アルゴリズム内の sort() を利用
    36   sort(wordSeq.begin(), wordSeq.end());
    37   cout << "整列後の単語列:" << endl;
    38   showContentsOf(wordSeq);
    39 }
    40
    41 void showContentsOf(vector<string> wordSeq) {
    42   int lineLength = 0;
    43   for (vector<string>::iterator p=wordSeq.begin(); p!=wordSeq.end(); ++p) {
    44     if (lineLength+p->size()+1 >70) {
    45       cout << endl;
    46       lineLength = 0;
    47     }
    48     cout << *p << " ";
    49     lineLength += p->size() + 1;
    50   }
    51   cout << endl;
    52 }
[motoki@x205a]$ g++ useSTL_algorithm.cpp
[motoki@x205a]$ cat useSTL_mapStringInt.data
C++, invented at Bell labs by Bjarne Stroustrup in the mid-1980s,
is a powerful modern successor language to C.  C++ adds to C the
concept of class, a mechanism for providing user-defined types,
also called abstract data types.  C++ supports object-oriented
programming by these means and by providing inheritance and
runtime type binding.
[motoki@x205a]$ ./a.out < useSTL_mapStringInt.data
入力単語列:
C++, invented at Bell labs by Bjarne Stroustrup in the mid-1980s, is
a powerful modern successor language to C. C++ adds to C the concept
of class, a mechanism for providing user-defined types, also called
abstract data types. C++ supports object-oriented programming by
these means and by providing inheritance and runtime type binding.
---
最初の"C++"以降の 10 単語列:
C++ adds to C the concept of class, a mechanism
---
整列後の単語列:
Bell Bjarne C C++ C++ C++, C. Stroustrup a a abstract adds also and
and at binding. by by by called class, concept data for in
inheritance invented is labs language means mechanism mid-1980s,
modern object-oriented of powerful programming providing providing
```

```
runtime successor supports the the these to to type types, types.
user-defined
[motoki@x205a]$
```

ここで、

- プログラム 18~24 行目 では、標準入力から単語を次々に入力し、vector<string>型オブジェクト wordSeq に順に記録している。
- プログラム 20 行目 に現れる push_back() は vector<string>型オブジェクトに備わったメンバ関数で、引数で指定されたデータをオブジェクトの最後尾に追加する働きをする。
- プログラム 28~33 行目 では、STL アルゴリズム内の find() を用いて wordSeq に保存された中から文字列"C++"を探し、その場所以降の 10 個の単語を表示している。実行結果においては、入力単語列冒頭に現れる"C++,"は"C++"とは異なる単語として扱われ、データファイル 2 行目の"C++"が見つけ出されている。
- プログラム 28 行目 に現れる find() は STL アルゴリズムのライブラリ algorithm に備わった関数で、第 1 引数で指定された場所から第 2 引数で指定された場所の 1 つ手前の場所の間の中から第 3 引数で指定されたデータを探し、見つかったらその要素へのポインタ (に相当するもの) を、見つからなかったら第 2 引数で指定されたポインタ (に相当するもの) を返す。
- プログラム 36~38 行目 では、STL アルゴリズム内の sort() を用いて wordSeq に保存された単語を辞書順に並び替え、更新後の wordSeq 内の単語を全て表示している。
- プログラム 36 行目 に現れる sort() は STL アルゴリズムのライブラリ algorithm に備わった関数で、第 1 引数で指定された場所から第 2 引数で指定された場所の 1 つ手前の場所の間のデータを昇順に並べ替える。

演習問題

□演習 6.1 (型パラメータ付きの関数定義) 型パラメータ付きの関数定義を用いることにより、演習 2.9 と同等のことを行う C++プログラムを作成せよ。

7 例外処理

- C 言語 assert() 関数を用いた例外処理
- C++言語における例外処理, 例外クラスのライブラリ

7.1 C 言語 assert() 関数を用いた例外処理

{Pohl(1999)9.1 節 }

一般に、プログラムや関数には正しく動作するための前提条件があり、既存のプログラムや関数を利用する際はそれぞれの前提条件を守った上での利用が必要である。ただ、プログラム外からの入力データに想定外のものが混じるここもあり得るし、また不注意によるプログラムの書き間違いもあり得る。注意深くしても前提条件が満たされなくなる状況を完全に除去するのは難しい。かと言って、前提条件が満たされないプログラムや関数を動作させても、資源の無駄使いにしかならない。そこで、C 言語では、assert() 関数を用いることによって、想定外の状況 (**例外**, exception, と呼ぶ) が起きてないか調べ、そういった状況を検出したらすぐに強制終了できる様になっている。

例 7.1 (C 言語 assert() を用いた例外処理, 添字チェックの機能を備えた配列のクラス) 例 6.2 で「添字の有効性をチェックする機能を備えた配列」のテンプレートクラス SafeArray を定義する際、既に assert() 関数を用いて例外発生の検出 (とその場合の強制終了) の措置を講じている。実際、ソースコードを再掲すると次の通り。(ここで、プログラム中の下線は assert() 関数を呼び出した箇所を表す。)

```
[motoki@x205a]$ cat -n SafeArray_verAssert.h
     1 /* 「添字の有効性をチェックする機能を備えた配列」                    */
     2 /* をインスタンスとする型パラメータ付きクラス SafeArray の仕様部   */
     3
     4 #ifndef ___Class_SafeArray
     5 #define ___Class_SafeArray
     6
     7 template<typename TYPE>
     8 class SafeArray {
     9   TYPE* basePtr;                  //pointer to the 1st element
    10   int size;                       //array size
    11 public:
    12   explicit SafeArray(int n=100);        //create a SafeArray of size n
    13   SafeArray(const SafeArray<TYPE>& a);  //copy constructor
    14   SafeArray(const TYPE a[], int n);     //copy a standard array
    15   ~SafeArray() { delete[] basePtr; }
    16   int getSize() { return size; }
    17   typedef TYPE* iterator;
    18   iterator begin() { return basePtr; }
    19   iterator end() { return basePtr + size; }
    20   TYPE& operator[](int i);              //range-checked element
```

```
21   SafeArray<TYPE>& operator=(const SafeArray<TYPE>& a);
22 };
23
24 //=========================================================================
25 /* 「添字の有効性をチェックする機能を備えた配列」                          */
26 /* をインスタンスとする型パラメータ付きクラス SafeArray の実装部   */
27
28 #include <cassert>
29
30 template<typename TYPE>        //create a SafeArray of size n
31 SafeArray<TYPE>::SafeArray(int n): size(n)
32 {
33   assert(n > 0);
34   basePtr = new TYPE[size];
35   assert(basePtr != 0);
36 }
37
38 template<typename TYPE>        //copy constructor
39 SafeArray<TYPE>::SafeArray(const SafeArray<TYPE>& a)
40 {
41   size = a.size;
42   basePtr = new TYPE[size];
43   assert(basePtr != 0);
44   for (int i=0; i<size; ++i)
45     basePtr[i] = a.basePtr[i];
46 }
47
48 template<typename TYPE>        //copy a standard array
49 SafeArray<TYPE>::SafeArray(const TYPE a[], int n)
50 {
51   assert(n > 0);
52   size = n;
53   basePtr = new TYPE[size];
54   assert(basePtr != 0);
55   for (int i=0; i<size; ++i)
56     basePtr[i] = a[i];
57 }
58
59 template<typename TYPE>        //range-checked element
60 TYPE& SafeArray<TYPE>::operator[](int i)
61 {
62   assert (0<=i && i<size);
63   return basePtr[i];
```

```
 64 }
 65
 66 template<typename TYPE>        //assignment operator
 67 SafeArray<TYPE>& SafeArray<TYPE>::operator=(const SafeArray<TYPE>& a)
 68 {
 69   assert (a.size == size);
 70   for (int i=0; i<size; ++i)
 71     basePtr[i] = a.basePtr[i];
 72   return *this;
 73 }
 74
 75 #endif
[motoki@x205a]$
```

これに関して、

- プログラムの 33 行目, 35 行目,43 行目,51 行目, 54 行目,62 行目,69 行目 で用いている関数assert() は、引数で与えられた条件が満たされないと強制終了を引き起こすC言語の標準ライブラリ関数である。そして、この関数を使うために 28 行目 で「#include <cassert>」としている。
- テンプレートクラス SafeArray が上の様に定義されていれば、例外発生によってプログラムは次の様な出力結果をもたらす。(ここで、プログラム中の下線は例外発生を引き起こす箇所を表す。)

```
[motoki@x205a]$ cat -n useSafeArray_verAssert_1.cpp
     1 // テンプレートクラス SafeArray を利用した時に例外が起きる例
     2
     3 #include <iostream>
     4 #include "SafeArray_verAssert.h"
     5 using namespace std;
     6
     7 int main()
     8 {
     9   SafeArray<int> a(100);
    10
    11   for (int i=0; i<a.getSize(); i++) {
    12     a[i] = i+10;
    13   }
    14   cout << "a = { " << a[0] << ", " << a[1] << ", " << a[2]
    15        << ", ..., " << a[99] << " }" << endl;
    16   cout << "---" << endl;
    17
    18   cout << "a = { " << a[0] << ", " << a[1] << ", " << a[2]
    19        << ", ..., " << a[100] << " }" << endl;
    20 }
[motoki@x205a]$ g++ useSafeArray_verAssert_1.cpp
```

```
[motoki@x205a]$ ./a.out
a = { 10, 11, 12, ..., 109 }
---
a.out: SafeArray_verAssert.h:62: TYPE& SafeArray<TYPE>::operator[](int)
                 [with TYPE = int]: Assertion ‘0<=i && i<size’ failed.
中止
[motoki@x205a]$ cat -n useSafeArray_verAssert_2.cpp
     1 // テンプレートクラス SafeArray を利用した時に例外が起きる例
     2
     3 #include <iostream>
     4 #include "SafeArray_verAssert.h"
     5 using namespace std;
     6
     7 int main()
     8 {
     9   SafeArray<int>* a;
    10
    11   a = new SafeArray<int>(100000000);
    12   for (int i=0; i<(*a).getSize(); i++) {
    13     (*a)[i] = i+10;
    14   }
    15   cout << "*a = { " << (*a)[0] << ", " << (*a)[1] << ", " << (*a)[2]
    16        << ", ..., " << (*a)[99999999] << " }" << endl;
    17   delete a;           // (*a).~SafeArray<int>(); を伴う
    18   cout << "---" << endl;
    19
    20   a = new SafeArray<int>(1000000000);
    21   for (int i=0; i<(*a).getSize(); i++) {
    22     (*a)[i] = i+10;
    23   }
    24   cout << "*a = { " << (*a)[0] << ", " << (*a)[1] << ", " << (*a)[2]
    25        << ", ..., " << (*a)[999999999] << " }" << endl;
    26   delete a;           // (*a).~SafeArray<int>(); を伴う
    27 }
[motoki@x205a]$ g++ useSafeArray_verAssert_2.cpp
[motoki@x205a]$ ./a.out
*a = { 10, 11, 12, ..., 100000009 }
---
terminate called after throwing an instance of ’std::bad_alloc’
  what():  std::bad_alloc
中止
[motoki@x205a]$ cat -n useSafeArray_verAssert_3.cpp
     1 // テンプレートクラス SafeArray を利用した時に例外が起きる例
```

```
 2
 3 #include <iostream>
 4 #include "SafeArray_verAssert.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9   SafeArray<int> a(100);
10   for (int i=0; i<a.getSize(); i++) {
11     a[i] = i+10;
12   }
13   cout << "a = { " << a[0] << ", " << a[1] << ", " << a[2]
14        << ", ..., " << a[99] << " }" << endl;
15   cout << "---" << endl;
16
17   SafeArray<int> b(-100);
18   for (int i=0; i<b.getSize(); i++) {
19     b[i] = i+10;
20   }
21   cout << "b = { " << b[0] << ", " << b[1] << ", " << b[2]
22        << ", ..., " << b[99] << " }" << endl;
23 }
```

```
[motoki@x205a]$ g++ useSafeArray_verAssert_3.cpp
[motoki@x205a]$ ./a.out
a = { 10, 11, 12, ..., 109 }
---
a.out: SafeArray_verAssert.h:33: SafeArray<TYPE>::SafeArray(int)
                          [with TYPE = int]: Assertion ‘n > 0’ failed.
中止
[motoki@x205a]$
```

ここで、

◇ プログラム useSafeArray_verAssert_2.cpp 17 行目,26 行目 の delete 演算では、変数 a がオブジェクトへのポインタなので a の指しているオブジェクトのデストラクタ (*a).~SafeArray<int>() が呼び出される。

◇ プログラム useSafeArray_verAssert_1.cpp, useSafeArray_verAssert_3.cpp の場合は、それぞれSafeArray_verAssert.h 62 行目,33 行目 の assert() 関数によって例外が検出され実行中止に至っている。一方、useSafeArray_verAssert_2.cpp の場合は、SafeArray_verAssert.h 34 行目 の new 演算の最中に例外が検出され (「std::bad_alloc」という例外の種類を表すオブジェクトが生成され、それに対する対処という形で) 実行中止に至っている。 (従って、SafeArray_verAssert.h 中の 35 行目,43 行目, 54 行目 の assert() 関数の呼び出しは実際にはほぼ無意味である。)

7.2 C++言語における例外処理, 例外クラスのライブラリ

{Pohl(1999)9.2 節,9.8 節, プログラミング I(2017)20.4 節, 柴田 (2014)8.1-8.4 節, シルト (2012)11.3-11.5 節}

- C 言語の下で 前節 7.1 の様に assert() 関数を使えば、想定外の例外的状況が起きてないか調べ、そういった状況を検出したらすぐに強制終了することはできる。しかし、例外的な状況を検出した場合、即座に「強制終了」で、プログラマ自身がその例外を処理する選択肢は用意されていない。

- かと言って、C 言語の下でassert() 関数を使わずに正しくて頑健な (i.e. どんな状況に対してもうまく動く) プログラムを作ろうとすると、エラー検出とその回復のコードがプログラムのあちこちに埋め込まれ、本来の処理の流れが見えなくなってしまう恐れがある。

  例えば、
  getchar 関数を使って正整数を 2 個入力してそれらを基に何らかの計算をするという場合は、正整数が正しく入力されたかどうかのチェックが必要である。 場合によっては、入力ミスに対して再入力を促すなどの処理を行うことになる。この様な、エラー検出／回復のコードをプログラム中に埋め込むと、往々にして本来の処理の流れが見え難くなってしまう。

- C++言語 では、本来の処理の流れの明瞭さを損なわずにエラー検出等を行うために、**例外処理**の機構が導入されている。すなわち、
  ① 想定外の状況になっていることが実行中に検出されると、そのことを明らかにするために**例外** (exception) と呼ばれるデータを生成し**送出** (throw) する。
  ② 例外データが送出されると、(必要なら関数呼出しの履歴を逆にたどる等もして) 例外を処理してくれるコードが探される。
  ③ 例外処理のコードが見つかればそれを実行する。 見つからなければ、C++言語のエラー処理ルーチン がエラー報告を出して実行を中止する。

例 7.2 (C++における例外処理, 添字チェックの機能を備えた配列のクラス) 例 7.1 で示したテンプレートクラス SafeArray の定義中の assert() 関数呼出し部を、C++言語の例外処理の機構に沿って書き換えると次の様になる。(ここでは、比較のため、元々あった assert() 関数呼出しはコメントアウトしただけにとどめ残している。)

```
[motoki@x205a]$ cat -n SafeArray_verThrow.h
     1 /* 「添字の有効性をチェックする機能を備えた配列」                    */
     2 /* をインスタンスとする型パラメータ付きクラス SafeArray の仕様部    */
     3
     4 #ifndef ___Class_SafeArray
     5 #define ___Class_SafeArray
     6
     7 template<typename TYPE>
     8 class SafeArray {
     9   TYPE* basePtr;                  //pointer to the 1st element
    10   int size;                       //array size
```

```
11 public:
12   //例外クラスの定義
13   class IndexOutOfBounds {
14   public:
15     int index;
16     IndexOutOfBounds(int index): index(index) {}
17   };
18   //関数群
19   explicit SafeArray(int n=100);        //create a SafeArray of size n
20   SafeArray(const SafeArray<TYPE>& a);  //copy constructor
21   SafeArray(const TYPE a[], int n);     //copy a standard array
22   ~SafeArray() { delete[] basePtr; }
23   int getSize() { return size; }
24   typedef TYPE* iterator;      //「using iterator = TYPE*;」ではダメ
25   iterator begin() { return basePtr; }
26   iterator end() { return basePtr + size; }
27   TYPE& operator[](int i);              //range-checked element
28   SafeArray<TYPE>& operator=(const SafeArray<TYPE>& a);
29 };
30
31 //=========================================================================
32 /* 「添字の有効性をチェックする機能を備えた配列」                          */
33 /* をインスタンスとする型パラメータ付きクラス SafeArray の実装部   */
34
35 //#include <cassert>
36
37 template<typename TYPE>        //create a SafeArray of size n
38 SafeArray<TYPE>::SafeArray(int n): size(n)
39 {
40   //assert(n > 0);
41   if (n <= 0)
42     throw "Invalid safeArray size was specified in a constructor.";
43   basePtr = new TYPE[size];  //失敗すると std::bad_alloc 例外を送出
44   //assert(basePtr != 0);
45 }
46
47 template<typename TYPE>        //copy constructor
48 SafeArray<TYPE>::SafeArray(const SafeArray<TYPE>& a)
49 {
50   size = a.size;
51   basePtr = new TYPE[size];  //失敗すると std::bad_alloc 例外を送出
52   //assert(basePtr != 0);
53   for (int i=0; i<size; ++i)
```

```
    54     basePtr[i] = a.basePtr[i];
    55 }
    56
    57 template<typename TYPE>        //copy a standard array
    58 SafeArray<TYPE>::SafeArray(const TYPE a[], int n)
    59 {
    60   //assert(n > 0);
    61   if (n <= 0)
    62     throw "Invalid safeArray size was specified in a copy constructor.";
    63   size = n;
    64   basePtr = new TYPE[size];  //失敗するとstd::bad_alloc例外を送出
    65   //assert(basePtr != 0);
    66   for (int i=0; i<size; ++i)
    67     basePtr[i] = a[i];
    68 }
    69
    70 template<typename TYPE>        //range-checked element
    71 TYPE& SafeArray<TYPE>::operator[](int i)
    72 {
    73   //assert (0<=i && i<size);
    74   if (i<0 || size<=i)
    75     throw IndexOutOfBounds(i);  //例外オブジェクトを送出させる
    76   return basePtr[i];
    77 }
    78
    79 template<typename TYPE>        //assignment operator
    80 SafeArray<TYPE>& SafeArray<TYPE>::operator=(const SafeArray<TYPE>& a)
    81 {
    82   //assert (a.size == size);
    83   if (a.size != size)
    84     throw "Assignment was tried between different size safeArray objects.";
    85   for (int i=0; i<size; ++i)
    86     basePtr[i] = a.basePtr[i];
    87   return *this;
    88 }
    89
    90 #endif
[motoki@x205a]$
```

これに関して、

- プログラムの 13~17 行目 は、「配列の添字の値が許容範囲を超えている」という例外状況を表すオブジェクトのクラス IndexOutOfBounds を定義している。ここでは、これらのオブジェクトは SafeArray クラスに付随したものと考え、SafeArray クラスの中で局所的に定義している。この定義中の 15 行目 で宣言された index は配列の添

字として指定された範囲外の値を保持するための要素を表す。

- プログラム 42 行目,62 行目,75 行目,84 行目 の throw 文は、①(例外検出等の理由で) プログラム実行の通常の流れを止め、②検出された例外を表すデータ (throw キーワードの右側に置かれた引数) を生成し、③その例外データを処理してもらうためのコードを探す、という動作を引き起こす。特に、
  - ◇ 75 行目 は、配列の添字の範囲が許容範囲を超えた場合の処置で、例外データとして SafeArray<TYPE>::IndexOutOfBounds 型オブジェクトを生成して送出している。
  - ◇ 42 行目,62 行目 は、SafeArray 型配列を新規に生成する際に引数で与えられた配列サイズ 0 以下であった場合の処置で、例外データとしてエラーメッセージ (char* 型文字列) を生成して送出している。
  - ◇ 84 行目 は、SafeArray 型配列間の「要素ごとの代入」演算を行う際に要素数が異なった場合の処置で、例外データとしてやはりエラーメッセージ (char*型文字列) を生成して送出している。

  ここでは例示しなかったが、例外データとして int 型を始め任意の型のデータを送出することが可能である。
- プログラム 43 行目,51 行目,64 行目 に現れる new 演算子は、メモリ確保に失敗すると std::bad_alloc 型の例外オブジェクトを生成して送出する。従って、44 行目,52 行目,65 行目 の assert() 関数呼び出しについては、throw 文等で置き換えることはせずに、単にコメントアウトだけで済ませている。

  補足：古い C++言語では、new 演算子はメモリ確保に失敗すると NULL を返していたらしい。また現在の C++言語でも、new の代わりに new(nothrow) を用いると、メモリ確保失敗時に例外オブジェクトを送出させずに NULL を演算結果として返すそうである。
- テンプレートクラス SafeArray が上の様に定義されていれば、例外処理のコードも組み込んだプログラムとして次の様なものを考えることができる。

```
[motoki@x205a]$ cat -n useSafeArray_verThrow.cpp
     1 // テンプレートクラス SafeArray を利用した時に例外が起きる例
     2
     3 #include <iostream>
     4 #include <cstdlib>      // for abort()
     5 #include "SafeArray_verThrow.h"
     6 using namespace std;
     7
     8 int main()
     9 {
    10   try {
    11     SafeArray<int> a(100);
    12
    13     for (int i=0; i<a.getSize(); i++) {
    14       a[i] = i+10;
    15     }
    16     cout << "a = { " << a[0] << ", " << a[1] << ", " << a[2]
```

```
17          << ", ..., " << a[99] << " }" << endl;
18    cout << "---" << endl;
19
20    cout << "a = { " << a[0] << ", " << a[1] << ", " << a[2]
21          << ", ..., " << a[100] << " }" << endl;
22  }
23  catch (const bad_alloc& x) {
24    cerr << "bad_alloc exception was thrown." << endl;
25    //abort();
26  }
27  catch (const SafeArray<int>::IndexOutOfBounds& e) {
28    cerr << "IndexOutOfBounds exception: index=" << e.index << endl;
29    //abort();
30  }
31  catch (const char* error) {
32    cerr << error << endl;
33    //abort();
34  }
35  catch ( ... ) {
36    cerr << "Some exception was caught." << endl;
37    //abort();
38  }
39  cout << "================================================" << endl;
40
41  try {
42    SafeArray<int> a(100000000);
43    for (int i=0; i<a.getSize(); i++) {
44      a[i] = i+10;
45    }
46    cout << "a = { " << a[0] << ", " << a[1] << ", " << a[2]
47          << ", ..., " << a[99999999] << " }" << endl;
48    cout << "---" << endl;
49
50    SafeArray<int> b(1000000000);
51    for (int i=0; i<b.getSize(); i++) {
52      b[i] = i+10;
53    }
54    cout << "b = { " << b[0] << ", " << b[1] << ", " << b[2]
55          << ", ..., " << b[999999999] << " }" << endl;
56  }
57  catch (const bad_alloc& x) {
58    cerr << "bad_alloc exception was thrown." << endl;
59    //abort();
```

```
 60   }
 61   catch (const SafeArray<int>::IndexOutOfBounds& e) {
 62     cerr << "IndexOutOfBounds exception: index=" << e.index << endl;
 63     //abort();
 64   }
 65   catch (const char* error) {
 66     cerr << error << endl;
 67     //abort();
 68   }
 69   catch ( ... ) {
 70     cerr << "Some exception was caught." << endl;
 71     //abort();
 72   }
 73   cout << "================================================" << endl;
 74
 75   try {
 76     SafeArray<int> a(100);
 77     for (int i=0; i<a.getSize(); i++) {
 78       a[i] = i+10;
 79     }
 80     cout << "a = { " << a[0] << ", " << a[1] << ", " << a[2]
 81          << ", ..., " << a[99] << " }" << endl;
 82     cout << "---" << endl;
 83
 84     SafeArray<int> b(-100);
 85     for (int i=0; i<b.getSize(); i++) {
 86       b[i] = i+10;
 87     }
 88     cout << "b = { " << b[0] << ", " << b[1] << ", " << b[2]
 89          << ", ..., " << b[99] << " }" << endl;
 90   }
 91   catch (const bad_alloc& x) {
 92     cerr << "bad_alloc exception was thrown." << endl;
 93     //abort();
 94   }
 95   catch (const SafeArray<int>::IndexOutOfBounds& e) {
 96     cerr << "IndexOutOfBounds exception: index=" << e.index << endl;
 97     //abort();
 98   }
 99   catch (const char* error) {
100     cerr << error << endl;
101     //abort();
102   }
```

```
  103   catch ( ... ) {
  104     cerr << "Some exception was caught." << endl;
  105     //abort();
  106   }
  107   cout << "================================================" << endl;
  108
  109   try {
  110     SafeArray<int> a(100), b(100), c(200);
  111     for (int i=0; i<a.getSize(); i++) {
  112       a[i] = i+10;
  113     }
  114     b = a;
  115     cout << "b = { " << b[0] << ", " << b[1] << ", " << b[2]
  116          << ", ..., " << b[99] << " }" << endl;
  117     cout << "---" << endl;
  118
  119     c = a;
  120     cout << "c = { " << c[0] << ", " << c[1] << ", " << c[2]
  121          << ", ..., " << c[99] << " }" << endl;
  122     cout << "---" << endl;
  123   }
  124   catch (const bad_alloc& x) {
  125     cerr << "bad_alloc exception was thrown." << endl;
  126     abort();
  127   }
  128   catch (const SafeArray<int>::IndexOutOfBounds& e) {
  129     cerr << "IndexOutOfBounds exception: index=" << e.index << endl;
  130     abort();
  131   }
  132   catch (const char* error) {
  133     cerr << error << endl;
  134     abort();
  135   }
  136   catch ( ... ) {
  137     cerr << "Some exception was caught." << endl;
  138     abort();
  139   }
  140 }
[motoki@x205a]$ g++ useSafeArray_verThrow.cpp
[motoki@x205a]$ ./a.out
a = { 10, 11, 12, ..., 109 }
---
IndexOutOfBounds exception: index=100
```

```
================================================
a = { 10, 11, 12, ..., 100000009 }
---
bad_alloc exception was thrown.
================================================
a = { 10, 11, 12, ..., 109 }
---
Invalid safeArray size was specified in a constructor.
================================================
b = { 10, 11, 12, ..., 109 }
---
Assignment was tried between different size safeArray objects.
中止
[motoki@x205a]$
```

ここで、

◇ プログラム 10~22 行目,41~56 行目,75~90 行目,109~123 行目 の、キーワード try に続くブロックは try **ブロック**と呼ばれるもので、例外データの送出を監視して捕捉する実行区間を指定するものである。すなわち、これらの区間内のコードを実行中に送出された例外データだけが例外処理の対象として扱われることになる。

◇ try ブロック直後の 23~38 行目,57~72 行目,91~106 行目,124~139 行目 中に現れる catch (···) {···} という形のコードは、それぞれ**例外ハンドラ**と呼ばれるものである。その中の、キーワード catch の直後の (···) の部分に処理可能な例外データの種類が記述され、それに続くブロック (catch **ブロック**と呼ぶ) の部分に例外処理の内容が記述される。一般に、例外ハンドラが複数並んでいた場合は上から順に適用可能性が調べられ、最初に適用可能な例外ハンドラを用いて捕捉した例外データが処理される。また、try ブロックの後方に適用可能な例外ハンドラがない場合は、try ブロックを含む関数の呼び出し元の関数内で捕捉した例外データを処理する例外ハンドラが探される。具体的に 23~38 行目の場合 は、捕捉された例外データを処理するコードを探す作業は次の順に進む。

① 23~26 行目の例外ハンドラ の適用可能性を判断するために、捕捉された例外データが bad_alloc 型と合致するかどうかが調べられる。そして、もし合致していれば、例外データに x という別名が付けられた上で 23~26 行目の catch ブロックが実行され (**関数呼び出しの際の引数結合と同じ要領**)、例外処理を終了する。 また、もし非合致なら、次に

② 27~30 行目の例外ハンドラ の適用可能性を判断するために、捕捉された例外データが SafeArray<int>::IndexOutOfBounds 型と合致するかどうかが調べられる。そして、もし合致していれば、例外データに e という別名が付けられた上で 27~30 行目の catch ブロックが実行され、例外処理を終了する。また、もし非合致なら、次に

③ 31~34 行目の例外ハンドラ の適用可能性を判断するために、捕捉された例外データが char* 型と合致するかどうかが調べられる。そして、もし合致していれば、例外データに error という別名が付けられた上で 31~34 行目の catch ブロックが実行され、例外処理を終了する。 また、もし非合致なら、次に

④ 35~38 行目の例外ハンドラ の適用可能性が調べられる。その結果、このハンドラの適用可能な範囲としてワイルドカードに相当する「...」が指定されているので、無条件で 35~38 行目の catch ブロックが実行され、例外処理を終了する。

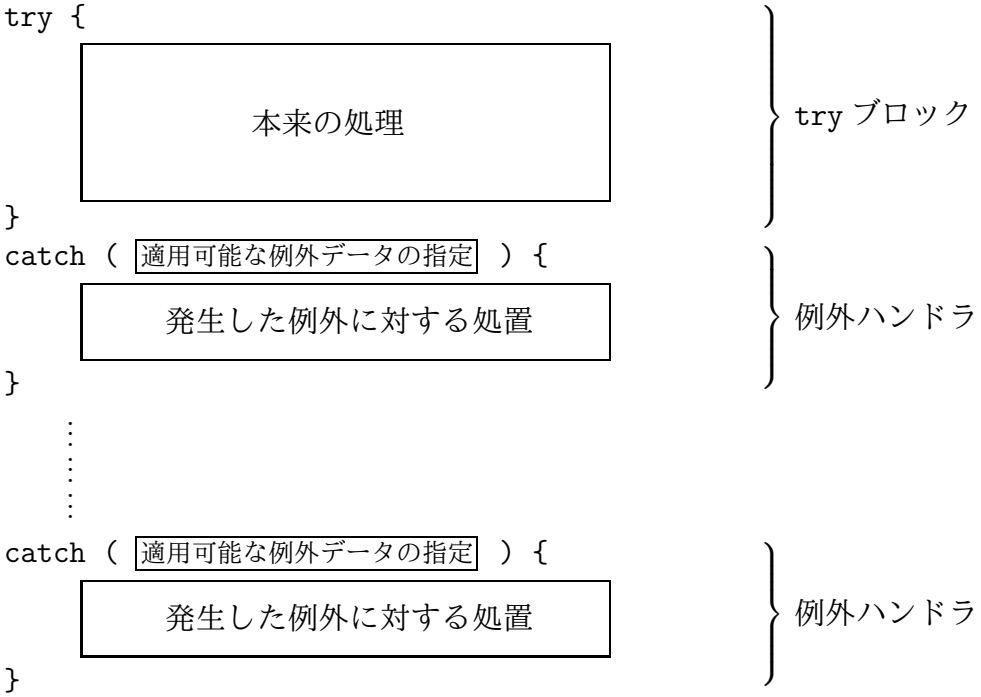
◇ プログラム 23~38 行目,57~72 行目,91~106 行目,124~139 行目 に置かれた例外ハンドラ群は全て同じものである。16 行から成る同じコードが 4 箇所に配置されているのを見て「それぞれの例外処理のコードを SafeArray テンプレートクラスの中に組み込んでしまえば良いのに」と思うかもしれない。確かに、この組み込みは可能で、これによって SafeArray クラスを利用するプログラムのコードは短くて済む。しかし、一般に定義したクラスは色々な用途に使う可能性があり、例外処理の内容も利用毎に異なる可能性があるので、ここでは SafeArray テンプレートクラスの定義中に try-chatch の構文を置くのを避け、クラスを利用するプログラム上で例外処理を行う様にしている。

◇ プログラム 24 行目,28 行目,32 行目,36 行目, 58 行目,62 行目,66 行目,70 行目, 92 行目,96 行目,100 行目,104 行目, 125 行目,129 行目,133 行目,137 行目 に現れる cerr は iostream ライブラリ内の標準ストリームで、標準エラー出力を表す。

◇ プログラム 126 行目,130 行目,134 行目,138 行目 に現れる abort() 関数はプログラムの強制終了 (異常終了) を引き起こす C 言語の標準ライブラリ関数である。そして、この関数を使うために 4 行目 で「#include <cstdlib>」としている。

◇ 例外を検出すると通常は強制終了するのであろうが、上のプログラムの場合は例外データを処理する 4 つの独立した例を示すのが目的なので、1~3 番目の例外処理の例の中の abort() 関数呼び出し (25 行目,29 行目,33 行目,37 行目, 59 行目,63 行目,67 行目,71 行目, 93 行目,97 行目,101 行目,105 行目) はコメントアウトしている。

C++言語における例外処理の機構 (まとめ)： 例外的状況の発生を考慮に入れない本来の処理の流れの見通しに悪影響を及ぼすことなく、例外処理を行いたい場合、C++言語では次の様な形のコードを書く。



ここで、

- try ブロックの中には、基本的には

  > 例外的状況の発生を考慮に入れない本来の処理

  を書き、更にその中 (もしくはそので呼ばれた関数処理の中) の所々に

  > 想定外の状況に陥ってないかの点検を行い、
  > もし想定外の状況になっていれば、①検出された例外を表すデータ (**例外データ**) を生成し、②それを throw 文を用いて**送出**するコード

  を挿入する。但し、例外データを生成し送出する際の throw **文**は次の形式で書く。

  ```
  throw 式;
  ```

- try ブロック直後に並んだ例外ハンドラは、try ブロック内で例外データが送出された場合のためのコードで、送出された例外データを分担して処理する。キーワード catch の直後の 適用可能な例外データの指定 の部分は例外ハンドラの対応可能な例外データの型を明示するもので、次の 3 つの形を基本として必要に応じて const 指定や参照宣言を加えた書き方ができる。

  ```
  データ型 変数名 (仮引数)
  データ型
  ...
  ```

  この内 1 番目の書き方は関数定義の仮引数の書き方と同じで、処理を受け持つことになった例外データについては、例外データと仮引数の結合が為された上で catch ブロック内の例外処理が施される。また、3 番目の書き方はワイルドカードを表し、これが指定された例外ハンドラは任意の例外データに適用される。

- catch (適用可能な例外データの指定) 直後の catch ブロックは、対応可能な例外データに対する処理内容を書く部分である。このブロックの中では、次の様に書いて処理中の例外データを**再送出** することもできる。

  ```
  throw;
  ```

- try ブロック以降の全体の処理の流れは、次の通り。

  ① まず、本来の処理の流れを記述した try ブロックの処理が試行される。 この中でエラー等が検出されると、それに相当する例外データが送出されtry ブロックの処理は中止される。

  ② try ブロック終了時点に例外データが発生していれば、try ブロック直後に置かれた例外ハンドラが上から順に調べられる。

  - ◇ もし、キーワード catch の直後の ( ) の中に送出された例外データの型と同じ (またはその基底クラスの) 型が指定されていたり、文字列「...」が指定されていたりした場合は、その例外ハンドラが例外への対処を担当する例外ハンドラとなり、送出された例外データを**捕捉** する。キーワード catch の直後の ( ) の中に仮引数も指定されていれば、例外データと仮引数の結合が為される。そして、catch ブロック内の例外処理が施される。(調べられずに残った例外ハンドラは無視される。)
  - ◇ 送出された例外データを捕捉する例外ハンドラが見つからない場合は、実行中の関数を呼び出した関数に対して例外データが送出される。

例外クラスのライブラリ： C++言語では、例外オブジェクトを表すためのクラスが標準ライブラリの中に用意されている。例えば次の通り。

exception ··· 全ての例外クラスの基底クラスで、ヘッダ<exception> で定義されている。このクラスの派生クラスとして次のクラスが用意されている。

- bad_exception ··· 例外指定に対する違反を表すオブジェクトのクラスで、ヘッダ<exception>で定義されている。
- bad_alloc ··· 記憶域確保の失敗を表すオブジェクトのクラスで、ヘッダ<new>で定義されている。
- bad_cast ··· 動的キャストの失敗を表すオブジェクトのクラスで、ヘッダ <typeinfo> で定義されている。
- bad_typeid ··· typeid 式中に空ポインタが含まれることを表すオブジェクトのクラスで、ヘッダ<typeinfo>で定義されている。
- logic_error ··· プログラム実行前に検出可能な**論理エラー**を表すオブジェクトのクラスで、ヘッダ<stdexcept> で定義されている。このクラスの派生クラスとして更に次のクラスが用意されている。
  - ◇ domain_error ··· ドメインエラー (<stdexcept>)
  - ◇ invalid_argument ··· 不正な実引数 (<stdexcept>)
  - ◇ length_error ··· 最大長を超えた長さのオブジェクト生成 (<stdexcept>)
  - ◇ out_of_range ··· 実引数の値が範囲外 (<stdexcept>)
- runtime_error ··· プログラム実行前に検出できない**実行時エラー**を表すオブジェクトのクラスで、ヘッダ<stdexcept>で定義されている。このクラスの派生クラスとして更に次のクラスが用意されている。
  - ◇ range_error ··· 内部計算で発生する範囲エラー (<stdexcept>)
  - ◇ overflow_error ··· 算術的なオーバーフローエラー (<stdexcept>)
  - ◇ underflow_error ··· 算術的なアンダーフローエラー (<stdexcept>)

演習問題

8 オブジェクト指向プログラミング (まとめ)

- オブジェクト指向言語の特徴とその恩恵
- オブジェクト指向言語におけるメモリ領域の使い方
- オブジェクト指向の設計
- ソフトウェアの部品化と再利用

8.1 オブジェクト指向言語の特徴とその恩恵

{Pohl(1999)10.1-2 節, プログラミング I(2017)22.16 節, Cardelli&Wegner(1985)1.3 節}

オブジェクト指向言語の特徴：

- **カプセル化** (encapsulation) と**情報隠蔽** (data hiding) ··· 関連したデータと操作を 1 つにまとめて (i.e. カプセル化して) 部品化し、外に見せる必要のないものは隠蔽できる。 (C++言語では、クラスを定義できるということ。⟶ 第 3.2 節)
- **新規データ型についての拡張** (type extensibility) ··· 新規にデータ型を追加して基本データ型と同じ様に簡単に使える。 (C++言語では、定義したクラスの名前をデータ型名として使うことができ、また「演算子の多重定義」を適切に行ったりすることにより、それらの型のデータに関する演算も基本データ型の場合と同じ様に簡単に使える様になる、ということ。⟶ 例 1.4, 例題 4.4)
- **継承** (inheritance) ··· 既存のデータ型を利用して新しいデータ型を定義できる。 (C++言語では、⟶ 第 5.1 節)
- **関数の動的結合による多態性** (polymorphism with dynamic binding) ··· 多態変数でオブジェクトを保持した時、保持したオブジェクトの種類に応じて動的に適切な関数 (i.e. オブジェクト自身のメンバー関数) を呼び出せる。 (C++言語では、⟶ 第 5.2 ∼5.4 節)

これらの特徴はどれも C++で新規に導入されたものではなく、C++言語以前の言語にも散見される。ただ、C++以前はソフトウェア業界内にオブジェクト指向の考え方を取り込む動きは少なかった。しかし、1980 年台に C++が出現すると、次の理由で大々的に C++が受け入れられ、オブジェクト指向の考え方が浸透していった。

◇ C 言語から C++言語への移行は容易、
◇ C++言語は処理効率の面でも実用的、
◇ カプセル化、情報隠蔽、継承、多態性、型安全、等の恩恵

C++言語の普及により、その後の Java 言語を始めとしたオブジェクト指向言語の普及にも繋がった。

抽象データ型、カプセル化、情報隠蔽： 関連したデータと操作を 1 つにまとめ (i.e. **カプセル化**し) たものは元々のコンピュータに備わった種類のデータではないが、ソフトウェア部品として扱いたいので、抽象的な型 (**抽象データ型**という) をもったデータと考える。

この種のソフトウェア部品は一人で作って利用するとは限らないので、ソフトウェア部品を提供する側の視点と利用する側の視点を区別することが大切である。

ソフトウェア部品の利用者の視点

- 利用し易いのが良い。
- 理解し易いのが良い。
- 安価で、効率的で、強力なのが良い。
- 内部の、利用上知る必要のない箇所が隠蔽されていれば、
  - ◇ その分、理解し易くなる。

ソフトウェア部品の提供者の視点

- 再利用や拡張を容易に行えるのが良い。
- 利用者が誤用しにくいのが良い。
- 内部を隠蔽できれば、
  - ◇ その部分を利用者に説明する必要がなくなる。
  - ◇ その部分は、(利用者にコード修正をお願いすることなく) 後で自由に改良を加えることができる。
  - ◇ 利用者の裏技的な誤使用の可能性を少なくすることができる。

⇨ **ソフトウェア部品利用者には、最低限必要なデータや操作だけを利用可能にし、残りは隠蔽するのが良い。**

ソフトウェア部品の再利用、継承： プログラミング言語が好んで使われるためには、ライブラリの生成やソフトウェア部品の再利用に関する性能も重要である。これに関しては、

- 既存のクラスを基に新規にクラスを派生定義する**継承を行えば**、既存クラスの**コード**が派生クラスと**共有**され、**再利用**されることになる。
- 互いに関連した多くの抽象データ型を継承によって**うまく整理して階層的に派生定義することが出来れば、コードの共有**も進み、また**インタフェースの統一**にも役立つ。
- 継承によって形成されるクラス階層において、派生クラスのオブジェクトを基底クラスのオブジェクトとして見ると、派生クラスの定義の際に追加した要素は隠れることになる。それゆえ、**クラス階層に沿った情報隠蔽も自然発生**していると考えることもできる。

多態性： 一般のプログラミング言語において、単一の要素 (e.g. 変数や関数, 定数, オブジェクト, 式) に単一の型のものしか対応させられない時にその要素 (に関する型の扱い) は**単態性** (**単相性**,monomorphism) を持つと言い、また単一の要素に対して複数の型のものへの対応付けが許される時は**多態性** (**多相性**,polymorphism) を持つと言う。また、多態性を実現するに当たって、1 つのコードで複数の型への対応付けが為される時に**普遍的多態性** (universal polymorphism, 純多態性, pure polymorphism) と呼び、見かけ上複数の型への対応付けが為されているが最終的な振舞いに至るまでの処置が型ごとに異なる時に**場当たり的多態性** (ad-hoc polymorphism) と呼ぶ。Cardelli&Wegner(ACM Computing Survey, vol.17, pp.471–522, 1985) に従うと、多態性は次の 4 つに分類される。

- **強制型変換** (coercion, 自動型変換) · · · 場当たり的多態性の一種。関数や演算子の処理において、コンパイラが与えられた実引数の型を見て必要に応じて型変換のコードを補うことによって、関数や演算子が複数のデータ型に対処できる様にする。

- **多重定義** (overloading)··· 場当たり的多態性の一種。実体として型ごとに個別に有限個の単態性要素をプログラマが用意し、コンパイラがこれらを切り替えて複数の型への対応付けを行う。⟶ 5.3 節, 2.11 節
- **サブタイプ多態性** (subtyping, 部分型付け, 包含,inclusion)··· 普遍的多態性の一種。多態変数でオブジェクトを保持した時、保持したオブジェクトの種類に応じて動的に適切な関数 (i.e. オブジェクト自身のメンバー関数) を呼び出せる、ということ。より具体的に C++言語では、基底クラス上で仮想関数 $f()$ が定義されていた場合、その基底クラスに属するオブジェクトへのポインタ変数 p は、その派生クラスに属するオブジェクトへのポインタを保持することもでき、「p->$f()$」と書いた部分は、コンパイラにより

  実行の時点で p が指しているオブジェクト内の $f()$ を動的に呼び出す

  様に翻訳される。⟶ 5.2 ∼5.4 節
- **パラメータ多態性** (parametric polymorphism)··· 普遍的多態性の一種。パラメータ付きクラス定義、総称的プログラミング。　例えば、テンプレートクラスが定義され、そのテンプレートを具現化したクラスをプログラム中に書いた時、コンパイラは具現化の指定に従って元々のテンプレートからパラメータなしのクラス定義を実際に創り出し、利用する。⟶ 第 6 節

オブジェクト指向の恩恵：

- クラスを定義し、そのクラスのインスタンス (ソフトウエア部品) を必要なだけ生成して利用する。

  ⟹ (恩恵 0) **コードの簡素化** ··· 類似コードをあちこちに書かなくて済むので。

以下、**オブジェクト指向の 3 大要素**

- **カプセル化** ··· 情報隠蔽を進めてオブジェクト (ソフトウェア部品) の独立性を高める。

  ⟹ (恩恵 1.1) **モジュール性** ··· 他のオブジェクトと切り離して、オブジェクト毎にソースコードを作成・保守することができる。

  (恩恵 1.2) **情報隠蔽の恩恵** ··· 内部の実装方式がちゃんと隠蔽されていて隠蔽しているはずの事柄に依存したコードが他のオブジェクト中に現れないことが保証されるなら、オブジェクト内部の実装方式を自由に変更することができ、他のオブジェクトとは独立に自由にオブジェクトの改良を進めることができる。

  もう少し具体的に言うと、 生情報を無神経に外部に向けて公開している場合は、ソフトウェア開発中にこの情報の保持方法を変更するとそれに伴ってこの情報を利用している全てのコードの修正が必要になってしまう。一方、情報隠蔽を行なっている場合は、情報の保持方法の変更は外部に公開されている関数の処理内容を変更するだけで済む。

  (恩恵 1.3) **コードの再利用** ··· 一般的な処理を行うオブジェクトの場合、他からの独立性を高めることにより、コード再利用の可能性が高まる。

  (恩恵 1.4) **ソフトウェア全体の保守の容易さ** ··· 1 つのオブジェクトで異常が発生した場合でも、そのオブジェクトを代替オブジェクトに差し替えたり、場合によってはそのオブジェクトを全体から切り離して残りの部分を運用したり (fail soft)、ということを行い易い。

- **継承** ··· 既存のクラスの内容を引き継いで新たな別のクラスを定義できる。
  $\Longrightarrow$ (恩恵 2.1) **効率的なプログラミング** ··· 簡単に既存クラスを拡張できる。また、類似クラスができそうな場合は、それらの共通部分を親クラスとして構成することにより、類似コードをあちこちに書かなくて済む。
  (恩恵 2.2) **間違いの可能性の減少** ··· 各種機能を整理して配置し、類似コードが多数に場所に分散するのを極力避けることが出来るので、コードの修正忘れも少なくなる。
- **多態性** ··· 同じメソッドに対してオブジェクトごとに異なる振る舞い。
  $\Longrightarrow$ (恩恵 3.1) **コードの簡素化** ··· 同種のインスタンスを統合的に扱えるので。
  (恩恵 3.2) **クラス利用の簡単化** ··· 強制型変換は暗黙に行われる。サブタイプ多態性では、クラス利用者は基底クラスの仮想関数で定められたインタフェースだけを理解していれば十分。また、多重定義により同じ目的の関数に別々の名前を付ける必要がなくなった。

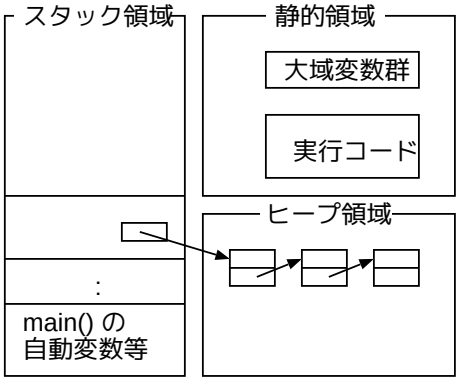
> 注意： ちゃんと書けばこういう恩恵に与れる、という話である。オブジェクト指向言語の場合は上の恩恵を引き出すための手立てが色々と用意されてはいるが、C++で書いても上の利点が保証される訳ではない。例えば、適切に情報隠蔽するかどうかはプログラマ次第である。
> 逆に、**以上の恩恵を充分に引き出せてないプログラムは C++で書いてあっても非オブジェクト指向**と言える。

8.2 オブジェクト指向言語におけるメモリ領域の使い方

{ 平澤 (2004)5 章, プログラミング I(2017)22.10 節 }

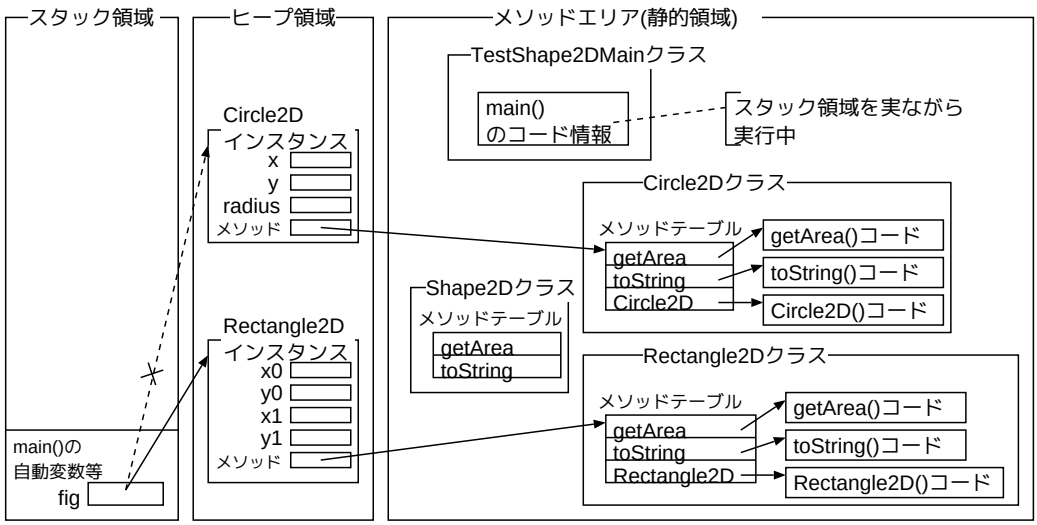
一般に、プログラム実行時にはメモリ領域は **静的領域、スタック領域、ヒープ領域** の 3 つに分けて管理される。

- 静的領域 ··· プログラムの実行コード、大域変数等を格納する領域。プログラムの実行時に確保され、終了するまでずっと内部の配置が固定される。
- スタック領域 ··· 関数呼び出しの柔軟な実現 (e.g. 再帰呼び出しを可能にする) のために利用される。
- ヒープ領域 ··· 動的なデータ記憶領域を確保したい時に利用される。特に C 言語では、`malloc()`, `realloc()`, `calloc()` といったライブラリ関数でメモリ領域確保を行い、`free()` でメモリ領域を解放する。



オブジェクト指向プログラムを実行する際のメモリの使い方について、基本的な枠組は上記の通りであるが、細部になると従来と異なる点もある。例えば、

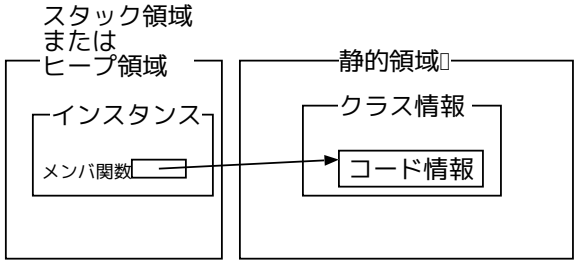
- 個々のクラス固有の情報 (e.g. どういう変数があるか, メソッドの実行コード) は静的領域に配置される。しかし、オブジェクト指向言語の種類 (e.g.Java, .NET) によっては、全てのクラス情報がプログラム開始時に一括してロードされるのではなく、必要になった時点でロードされる。そのため、例えば Java 言語 (1991 頃 ∼) ではクラス情報を格納する領域はもはや静的ではなく、メソッドエリアと呼ばれている。
- 従来は、動的データ構造を意識して使わない限り、ヒープ領域を使うことはあまりなかった。しかし、Java を始めとする最近のオブジェクト指向言語の場合は、インスタンスオブジェクトは全て動的に生成されるので、ヒープ領域内に配置される。それゆえ、ヒープ領域は頻繁に利用される。例えば、例題 5.7 で与えた C++ プログラムに相当するものを Java 言語で書いたとして、その中の、例題 5.7 最後,`useDerivedClassesFromShape2D.cpp` の 18 行目に相当する箇所を実行直後には、メモリ内部の状態は次の様になっているものと考えられる。



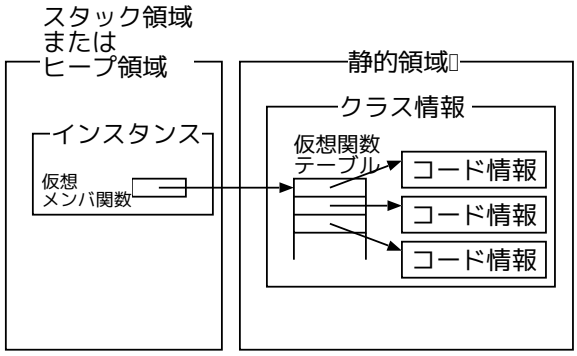
特に C++ 言語の場合、

- 全てのクラス情報はプログラム開始時に静的領域に一括してロードされる。
- C++言語 (1979 年頃 ∼) を設計するに当たって、C 言語のシステムプログラミングに適した性格は重要視された。そのため、否応なく全てのインスタンスオブジェクトをヒープ領域に配置するということには至らず、インスタンスをスタック領域に配置する選択肢も残された。
- インスタンスをヒープ領域に配置するためには、空き領域演算子 `new` を用いてインスタンスのための領域をヒープ領域内に確保し、そこにインスタンスを構成する。確保された領域へのポインタが `new` 演算の結果として得られるので、そのポインタ値をスタック領域内の変数に保持することになる。(ヒープ領域内へのポインタを保持する変数を宣言したブロックを出る際は、当然、(必要ならデストラクタを呼び出し、その後で)`delete` 演算子を呼び出して確保していた領域を開放する。)
- インスタンスの構成メンバとして関数が設定されていた場合でも、メモリ節約のため、当然のことながら確保されたインスタンス領域の中にメンバ関数のコード情報が格納されることはない。メンバ関数のコード情報は、クラス情報の一部として静的領

域内に配置され、そこへのポインタが個々のインスタンス領域の中に置かれる。



また、多態性の実装を容易に行うために、クラス情報の領域内には、クラス内で用意された仮想関数 (のコード情報) へのポインタを要素とする配列 (**仮想関数テーブル**と言う) が配置され、個々のインスタンスの中にはその仮想関数テーブルへの参照情報だけが (仮想メンバ関数のコード情報群の代わりに) 置かれることもある。



8.3 オブジェクト指向の設計

{Pohl(1999)10.1.2 節,10.3 節,
日経ソフトウェア編 (2009) 第 3 部 1 章,
日経ソフトウェア編 (2011) 第 2 部 2 章}

オブジェクト指向ソフトウェア設計の流れ： {Pohl(1999)10.1.2 節 }

クラス継承／階層はソフトウェア全体の設計に影響を及ぼすもので、システム構築やコード再利用に関わる重要箇所を含んだソフトウェアの骨格がそれによって決まる。それゆえ、オブジェクト指向の考えの下ではソフトウェア設計は次の流れで進む。

① 必要な抽象データ型群 (i.e. クラス群) を特定する。
② 前ステップで特定したクラスの間の関係を精査し、クラス継承によりコードとインタフェースの共通化を進める。
③ 仮想関数を用いて、必要に応じてオブジェクトに多態的な動きをさせる様にする。

ソフトウェア設計／クラス設計の際に考慮すべきこと： {Pohl(1999)10.3 節 }

互いに相反することもある次の事柄を考慮し、バランスの良い設計にする。

- **完全性** (completeness) ··· クラス設計の場合は、用意されたデータメンバとメンバ関数を使って行いたいことを全て行えるか、ということ。ブール代数においては、nand 演算だけで全てのブール式を表せるので、演算の集合 {nand} は完全であるが、通常、基本的な演算集合として {not, and, or} を用いている。この例の示す様に、完全性だけでは不十分なことが多く、実用のために十分な表現力が必要である。

- **表現力** (expressiveness) ··· クラス設計の場合は、十分なデータメンバとメンバ関数が用意されているか、ということ。
- **可逆性** (invertibility) ··· クラス設計の場合、用意されたどの主要メンバ関数に対しても逆の働きをするメンバ関数が用意されている、ということ。数値データ型の場合、加算と減算は互いに逆演算の関係にある。テキスト編集の場合、undo が全ての操作の逆操作に相当する。
- **直交性** (orthogonality) ··· クラス設計の場合は、生成されたインスタンスを利用する際に利用目的のために必要な操作系列が一意に決まる (i.e. 冗長性がない)、ということ。例えば平面上で図形を操作する場合、互いに直交した操作集合として { 水平移動, 垂直移動, 回転 } を考えることができる。
- **オッカムの剃刀** (Occam's Razor) ··· 元々は「ある事柄を説明するためには、必要以上に多くを仮定するべきでない」という指針で、より一般には「不要なものを (剃刀で) 切り落とす」ということ。クラス設計の際に継承によりコード共有を進めるのはこの指針に沿ったものと言えなくもない。
- **無矛盾性** (consistency) ···
- **単純性** (simplicity) ···
- **効率性** (efficiency) ···

オブジェクト指向設計の原則: { 日経ソフトウェア編 (2009)「ゼロから...」第 3 部 1 章 p.181}

- **単一責務の原則** (SRP, Single Responsibility Principle) ··· 分割された個々のプログラムに複数の目的・機能を負わせるべきでない、という指針。(その方が、将来の部分的な変更もやり易い。)
- **開放閉鎖の原則** (OCP, Open-Closed Principle) ··· 構築するクラス群は、機能拡張可能 (open) で、拡張の際には既存コードには手を加えなくて済む (closed)、様なものが良い、という指針。

良いプログラムを構築するための考え方と実践方法:

{ 日経ソフトウェア編 (2011)「Java ツール完全理解」第 2 部 2 章 }

- **ソフトウェアの価値の 3 条件**
  - ◇ **シンプル** ··· プログラムの理解や修正が容易になる。
  - ◇ **コミュニケーション可** ··· プログラムの書き手と読み手がソースコードを通じて十分にコミュニケーションできる。
  - ◇ **柔軟性** ··· 変更に対する柔軟性がある。(初期開発費用より修正費用の方が大きいので、これも重要。)
- **プログラミングの原則**
  - ◇ **YAGNI** (You Aren't Going to Need It.) ··· 今必要なことだけをやる。
  - ◇ **DRY** (Don't Repeat Yourself.) ··· コードの重複を避け、必要があればできるだけ再利用する。
  - ◇ **PIE** (Program Intently and Expressively.)··· 意図が明確に伝わる様にコードを書く。
- **代表的なベストプラクティス**

◇ **リファクタリング** · · · 外部に対する振舞いを変えずに、ソースコードの内部構造を整理し簡素化すること。
◇ **テストファースト** · · · 実装者の視点で実装コードを書く前に、利用者の視点でテストコードを書く。これによって余分な複雑さを排除できることを期待する。
◇ **ドメイン駆動設計** (Domain-Driven Design, DDD) · · · 問題領域 (domain) をモデル化し、それを中心に据えてソフトウェアを設計する。(モデル自体もドメイン知識を使って反復的に深化させていく。)

8.4 ソフトウェアの部品化と再利用

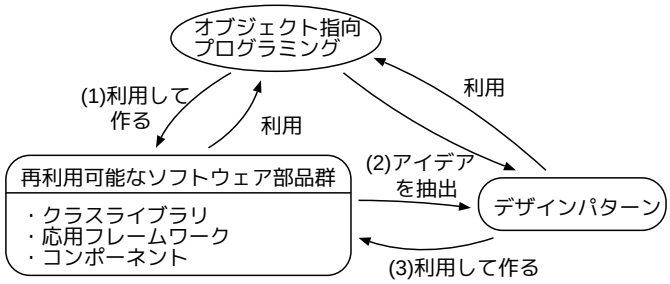
{平澤 (2004)3~4 章,6 章, 戸松 (1998)4~5 章, 結城 (2004)1 章, Pohl(1999)10.5 節}

プログラミング言語の進化 プログラミング言語は、単にアルゴリズムを記述できれば良いというものではない。アルゴリズムを容易にコード化できるための表現能力がまず必要である。また、ソフトウェアの寿命が伸びた結果、その保守も大事で、出来上がったプログラムの理解や修正の容易さも重要になっている。ソフトウェアの高い品質を得るために、プログラムの中に余計な複雑さや単純な間違いが入り込みにくいということも大切である。更に、ソフトウェアの生産性を上げるために、実績のあるプログラムを再利用できることも望ましい。これらの観点 (表現能力, 保守性, 品質保証, 再利用性) に照らしてプログラミング言語がどの様に進化してきたのかを、次の様にまとめることができる。

	表現能力	保守性	品質保証	再利用性	導入された機構/考え方
機械語					
アセンブリ言語	△				
高級言語	○			△	• サブルーチン (関数, 手続き) ⟶ 十分な表現能力, 再利用性向上
構造化	○	△	△	△	• goto 文を使わず処理手順を 3 つの基本構造 (順次進行, 条件分岐, 繰返し) だけで記述する ⟶ 保守性向上 • サブルーチンの独立性を高める ⟶ 再利用性多少向上
オブジェクト指向	○	○	○	○	• クラス (関連する変数とメソッドをまとめてソフトウェア部品を作る仕掛け) ⟶ 保守性向上, 再利用性向上 • 例外機構 ⟶ 品質向上 • 型チェックの強化 ⟶ 品質向上 • 多態性 ⟶ 再利用性向上 • 継承 ⟶ 再利用性促進

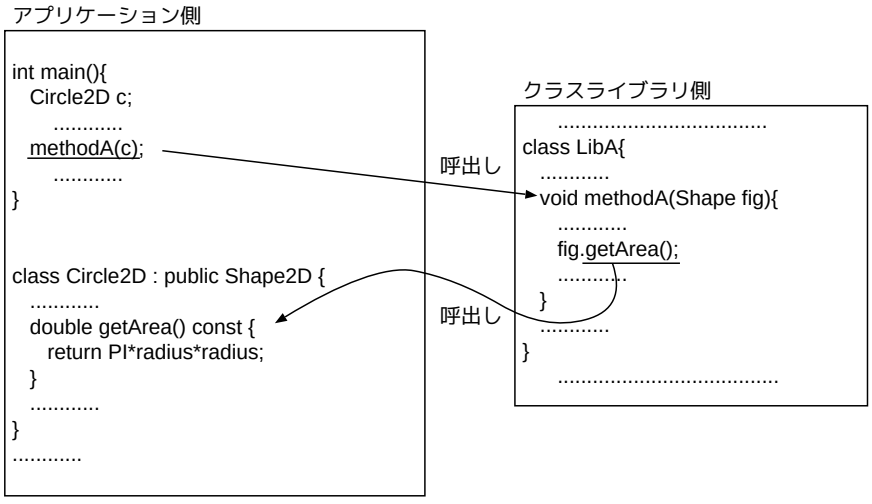
↓時間

ソフトウェア再利用技術発展の流れ オブジェクト指向より前の構造化言語では、再利用できるソフトウェアと言えばサブルーチン (関数, 手続き) だけで、コード変換や入出力処理, 数値計算, 文字列処理等の汎用ライブラリが提供される程度だった。しかし、オブジェクト指向の考えが導入され、関連性の強いサブルーチンや大域変数を 1 つのクラスとしてまとめて粒度の大きいソフトウェア部品を作り出す仕組みができると、ソフトウェア再利用の可能性は大きく広がった。その結果、C++や Java や.NET 環境でアプリケーションを開発する場合、再利用可能なソフトウェア部品として既に存在しているソースコードや実行形式モジュールを使い回すのが当り前になっている様です。再利用可能なソフトウェア部品としては、現在、**クラスライブラリ**、**(応用) フレームワーク**、**コンポーネント**と呼ばれるものがある。更には、ソフトウェア設計のアイデアを後で利用できるように文書化した、**デザインパターン**と呼ばれるものもある。蓄積されたデザインパターンを有効に活用できる様になれば、新たなソフトウェア部品の開発も容易になる。



クラスライブラリ その名の通り、汎用的な機能をもつクラスを多数蓄積したもののことです。オブジェクト指向の前と比べてソフトウェアが格段に豊富になったのに加え、ライブラリの利用の仕方も広がった。すなわち、関数ライブラリの場合は用意されたライブラリ関数をアプリケーションから呼び出すという利用の仕方だけだったが、クラスライブラリの場合は次の様な 3 つの利用の仕方が可能である。

- 用意されたクラスのインスタンスを作成して付属のメンバ関数等を利用。(クラスの利用, 従来のライブラリ関数呼び出しに相当。)
- ライブラリのコード内で実行されるメンバ関数として、アプリケーション固有の処理を呼び出す。 (多態性の利用。)



- ライブラリ内に用意されたクラスを拡張・補正して、新しいクラスを作成。 (継承の

利用。)

例えば、2011 年時点の Java(J2SE7.0; JDK1.7, Java SE Development Kit 1.7) には、GUI, 入出力, ネットワーキング, ... 等のために、合わせて 4000 にものぼるクラス (とインタフェースの) ライブラリが整備されていた。(オブジェクト指向言語では、言語仕様は最小限に抑えて必要な機能はクラスライブラリとして提供されるのが一般的で、これにより、言語仕様の互換性を保ちながらクラスライブラリの拡張によってバージョンアップを行うことが可能になる。)

$\Longrightarrow$ この様な豊富なライブラリを使いこなすことが、オブジェクト指向言語習熟への 1 つの道である。

(応用) フレームワーク 用途 (ドメイン) が類似したソフトウェアは多くの共通要素を持つので、アプリケーションの基本的な制御の流れをもったソフトウェアの枠組み (framework) を予め用意し、個別の処理のためにそこに将来組み込む予定のクラスやライブラリを明らかにしておけば、利用者の個別の要求のある部分だけを追加するだけで色々な利用者のニーズに合った応用プログラムを作成できる。こういった考えで作られた、特定の目的を果たすための「半完成品」の応用プログラムのことを**応用フレームワーク** (**アプリケーションフレームワーク**, application framework)、あるいは単に**フレームワーク**という。(クラスライブラリも応用フレームワークも再利用可能なソフトウェア部品群という点では同じである。ただ、目的と再利用部品の使われ方が違う。)

コンポーネント クラスライブラリと応用フレームワークは似ており、それらの境目はあいまいである。しかし、再利用可能なソフトウェア部品の種類として 3 番目に挙げる「コンポーネント」は他の 2 つとははっきりと違っている。平澤 (2004) によれば、

- クラスよりも粒度が大きく、
- (ソースコード形式でなく) バイナリ形式で提供される、

そして、

- ソフトウェア部品の定義情報も提供される、
- 機能的に独立性が高く内部の詳細を知らなくても利用できる、

というものを一般に**コンポーネント** (component) と呼んでいる様です。(広く浸透している定義ではない。) これらのソフトウェアの利用の仕方も特徴的で、ソースコードを書くのではなく、視覚的なツールを用いて直接関連する部品 (インスタンス) を配置・設定したり接続したりすることによって、短時間で応用ソフトウェアを組み立てる。最初のコンポーネント技術はマイクロソフト社の開発ツールである VisualBasic(1990 年代前半 ~) で導入されたもので、この技術は後に拡張されActiveX へとつながっている。一方、Java 環境においても、コンポーネント技術として **JavaBeans** と呼ばれる仕組みが用意されており、**Beans** と呼ばれるコンポーネントを BDK (JavaBeans Develpoment Kit) 等の (ビルダ) ツールで接続してソフトウェアを組み立てられる様になっている。

デザインパターン オブジェクト指向に基づいて再利用性や柔軟性の高いソフトウェアを開発しようとすると、様々な場面で何度もお決まりの設計指針が適用される。これらの「お決まりの設計指針」を記述したドキュメントを**デザインパターン** (design pattern) と言う。有意義で適用範囲の広いデザインパターンを見つけることが出来れば、①それ以降

の全てのソフトウェア開発者がその恩恵を受けることができ、また②それが開発者間の共通認識として定着すれば開発者間のコミュニケーションも容易になる。[この様な状況はアルゴリズムやデータ構造の場合と同じである。アルゴリズムやデータ構造のパターンについては (「アルゴリズムパターン」といった用語は使われないが) 既に多くの研究がなされており、有用なアルゴリズムは教科書で紹介されるなどして広く普及している。] 具体的なデザインパターンとしては、E.Gamma, R.Helm, R.Johnson, J.Vlissides という 4 人の技術者達 (**GoF** , the Gang of Four, と呼ばれている) が発表した次の 23 種類 (**GoF のデザインパターン**と呼ばれている) が有名で、これらは Java 言語のクラスライブラリの作成にも大いに利用されている。

GoF による分類	パターン名	再利用を妨げる要因のどれに効果が期待されるか?						
		クラス名を固定したインスタンスの生成	特定の処理内容への依存	プラットフォームに依存した application program interface の使用	特定のアルゴリズムへの依存	クラス同士の密接な依存関係	継承によるデメリット	クラス数の急激な増加
生成に関するパターン	Abstract Factory	○		○		○		
	Builder				○			
	Factory Method	○						
	Prototype	○						
	Singleton							
構造に関するパターン	Adapter							
	Bridge			○		○	○	○
	Composite						○	
	Decorator						○	○
	Facade					○		
	Flyweight							
	Proxy							
振舞いに関するパターン	Chain of Respon-sibility		○			○	○	
	Command		○			○		
	Interpreter							
	Iterator				○			
	Mediator					○		
	Memento							
	Observer					○	○	
	State							
	Strategy				○		○	
	Template Method				○			
	Visitor				○			

GoF の著作においては、これら 23 種類のデザインパターンは次の様な項目に分けて説明されている。

- パターン名 ··· デザインパターンの名前。
- 目的 ··· そのデザインパターンがどの様な設計課題に対処するか、何をもたらすか、原理と意図、等を簡潔に。

- （別名 · · · よく知られた別の名前があれば、それを。）
- 動機 · · · 設計上の問題点、及び、そのデザインパターン内のクラスやオブジェクトの構造がどの様にその問題を解決するか、を具体的に記述したシナリオ。
- 適用場面 · · · このデザインパターンを適用できる状況。
- 構造 · · · そのデザインパターン内で使われるクラス間の関係を **UML** (unified modeling language) のクラス図風に表現したもの、及び、要求のシーケンスやオブジェクト間の協調関係を図で表したもの。
- 構成要素 · · · そのデザインパターンで使われるクラス、オブジェクトや、それら各々の役割。
- 協調関係 · · · 各構成要素がどの様に協調して役割を果たすか。
- 結果 · · · そのパターンが要求に対してどの様に効果を発揮するか。
- 実装 · · · 実装する際に注意すべき落し穴、ヒント、技法や、言語に依存した問題について。
- サンプルコード · · · そのデザインパターンを使って実装した例。
- 利用例 · · · そのデザインパターンが実際のシステムで利用された例。
- 関連するパターン · · · 関係の深い別のデザインパターンとの関係。

例 8.1 (Iterator パターンの活用) 例えば、大きさ 10 の配列 `arr[]` の個々の要素に対してメンバ関数 `getName()` の実行依頼を出し、戻って来た文字列を全て表示するのに

```
for (int i=0; i<10; i++)
    cout << arr[i].getName() << endl;
```

と書いたのでは、「オブジェクトの集合体を表すのに配列を用いる」ということに依存したコードになってしまう。一方、データを `vector<`要素の型`>` 等の STL コンテナ `v` に入れ、Iterator パターンの指針に従えば、上の 2 行は

```
for (vector<要素の型>::iterator p=v.begin(); p!=v.end(); p++)
    cout << p->getName() << endl;
```

という風に、集合体の実装方法に依存しないコードに置き換わる。

例 8.2 (Singleton パターンの活用) インスタンス生成を 1 度だけに限定したいクラスもある。こういう場合のための方策として示されているのが **Singleton パターン**である。

例 8.3 (Strategy パターンの活用) 処理の大枠は固定するが、その中で使うアルゴリズム (strategy) は色々と切り替えて使いたい、という場合もある。こういう場合のための方策として示されているのが **Strategy パターン**である。

演習問題

索 引

記　号

A

B

C

D

E

F

さ 行

た 行

な 行

は 行

ま 行

ら 行