1. C++ (1) 「C から C++ へ」

♣ ここでは, 取り敢えず C++ でプログラムが書けることを目標に, C 言語との違いを中心に説明する.

1.1 取り敢えず動かしてみる

- 1 から n までの和を求める C++ プログラムの例

  行中の // より右にある文字はコメントであり, コンパイラはこれを無視する.

[List 1.1]

```
 1:  #include <iostream>  // C の <stdin.h> に相当
 2:
 3:  int main(void) {
 4:    std::cout << "hello" << std::endl;  // 出力の構文
 5:    int n;  // 宣言は, 使う前ならプログラム中のどこにあってもよい
 6:    std::cout << "n = ";
 7:    std::cin >> n;  // 入力の構文
 8:    int s = 0;
 9:    for (int i=1; i<=n; i++) s += i;
10:    std::cout << "sum(1.." << n << ") = " << s << std::endl;
11:    return 0;
12:  }
```

  – 3: 出力は << 演算で行なう. std::cout は標準出力で, C の stdout に相当する. (ちなみに C の stderr (エラー出力) は, C++ では std::cerr になる.) << を用いた構文では, 複数の項目を連続して出力することができる.

    std::endl は改行を表す.

    ```
    std::cout << "hello\n";
    ```

    と

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

    は, ほぼ同じ意味で, 大抵の場合どちらを書いても問題ないが, std::endl を推奨する. 厳密には, "\n" は改行記号のみを出力するのに対し, std::endl は改行記号の出力と同時に出力バッファのフラッシュを行う点が異なる.

  – 7: 入力は >> 演算を用いて行なう. std::cin は標準入力で, C 言語の stdin に相当する. 複数の >> 演算を用いれば, 連続した入力を行うことも可能.

  – 9: for 文の中で int i のような宣言を行うことも可能. この場合 i はこの for 文の中でだけで有効になる.

- コンパイルと実行

  C をコンパイルする gcc を g++ に変えるだけ. 上のプログラムを hello.cpp に格納したとすると,

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

  でコンパイルができる. C と同様, Cygwin の場合には a.exe, Linux や Mac OS X の場合は a.out という実行可能ファイルができるので,

  ```
  ./a.exe      (Cygwin の場合; ".exe" は省略可能)
  ```

  ```
  ./a.out      (Linux や Mac OS X の場合)
  ```

  でそれぞれ実行できる. これ以外の名前の実行ファイルを作成したい場合は, -o オプションで出力ファイル名を指定する. 例えば, myprog という名前を付けたいとき (Cygwin の場合は myprog.exe) は,

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

  のように指定すればよい.

- フォーマットつきの入出力

  数値の桁数を指定した出力もできるが, ここでは省略する. (取り敢えず C の printf はそのまま使える.)

課題 1.1 [List 1.1] のプログラムを入力し (コメントは省略して良い), コンパイル, 実行した後, プログラムに適当な修正を加え, 何か好きなものを計算するプログラムを作成せよ. (ドルの金額を入力して円に換算するとか, インチをセンチに換算するとか, 身長を入力して標準体重を出力する等, 何でも良いので各自作成せよ. ただし, 小数が必要な場合には, int 型ではなく double 型を用いること. どのようなプログラムを作成したかの説明, 実行結果, 実験状況・感想とともに, 作成したプログラムを提出すること.

いちいち "std::" を書くのが面倒

何かと std:: を書かなければならないので, 面倒と感じる場合は, プログラムの冒頭で

```
using namespace std;
```

と書けば省略できる. 推奨はしません. まあ, 名前が長いとかストローク数が多い (とかエラーメッセージが長い) のは, 取り敢えず C++ の文化と思ってしばらくつき合って下さい.

1.2 new と delete による変数領域の動的割当て

- C 言語で malloc(), free() で行っていた動的割当ては, new と delete で行う.

[List 1.2]

```
 1:  #include <iostream>
 2:
 3:  int main(void) {
 4:    int *a = 0;  // ポインタの宣言
 5:    a = new int;  // int 1 個分の領域を割り当て, そこへ のポインタを返す
 6:    *a = 100;
 7:    std::cout << *a << std::endl;
 8:    delete a;  // 領域の解放
 9:    return 0;
10:  }
```

- 4: 何も指さないポインタは, C では NULL で表したが, C++ では 0 で表すのが慣例となっている.
- 5: new による割当てが失敗した場合, 何も指定がなければプログラムはその時点で強制的に終了する. 正確には, 例外 (bad_alloc 例外) による割り込みが発生し, その例外に対する処理ルーチンが書いてあればそれが実行され, なければ強制終了ということになる. 例外処理は, catch, throw などの構文を用いて書くが, その詳細は少し難しいので省略する.

- 配列の動的割当てと解放は, それぞれ"new 型 [サイズ]" および"delete [] 変数名" という構文で行う. 次のプログラムでは, n に整数値を読み込み, サイズ n の整数配列を割り当てている.

[List 1.3]

```
 1:  #include <iostream>
 2:
 3:  int main(void) {
 4:    int n;
 5:    std::cin >> n;
 6:    int* a = new int[n];  // 整数 n 個分の領域を割り当て,
 7:                          // その先頭へのポインタを返す.
 7:    for (int i = 0; i<n; i++) a[i] = i;
 8:    delete [] a;  // delete の後の [] を 忘れないこと (忘れると悲惨!)
 9:    return 0;
10:  }
```

メモリーリーク

new で割り当てた領域は, 必ず delete で解放されるようにプログラムを書くのが鉄則である. これを忘れた場合, 使用しない記憶領域が解放されず, 記憶領域の再利用が行えなくなる. 小規模な例題に対する実行ではエラーは全く出ないので, 一見問題が無いように見えるが, 大規模な例に対して実行を行うと, メモリー不足が生じて実行が不可能になる. このようなバグをメモリーリーク (memory leak, memory leakage) と呼ぶ.

1.3 関数

1.3.1 引数の値渡しと参照渡し

- C 言語では, 関数呼び出しの際の引数の渡し方は「値渡し (call-by-value)」であった. C++ でも標準の引数の渡し方は, 値渡しである.

[List 1.4]

```
 1:  #include <iostream>
 2:
 3:  void inc(int x) {  // 整数値 x を受取って,
 4:    x ++;  // それを 1 増やして,
 5:    std::cout << "x = " << x << std::endl; //表示する
 6:  }
 7:
 8:  int main(void) {
 9:    int a = 5;
10:    inc(a);  // a を引数にして inc を呼び出すと,
11:    std::cout << "a = " << a << std::endl;  // a の値は変化するか?
12:    return 0;
13:  }
```

の実行結果は,

```
x = 6
a = 5
```

となり, a の値は変化しない.

- a の値を変更するような関数を書きたい場合, C 言語ではポインタを用いて次のように書いた.

[List 1.5]

```
 1:  #include <iostream>
 2:
 3:  void inc(int* x) {
 4:    (*x)++;
 5:    std::cout << "*x = " << *x << std::endl;
 6:  }
 7:
 8:  int main(void) {
 9:    int a = 5;
10:    inc(&a);
11:    std::cout << "a = " << a << std::endl;
12:    return 0;
13:  }
```

この実行の結果は,

```
*x = 6
a = 6
```

となる.

- C++ でもこれと全く同じ書き方ができるが, これに加えて「参照渡し (call-by-reference)」という構文が用意されている. 関数宣言の引数リスト中, 型の後に & をつけると, その変数への参照 (実態はアドレス) が渡される. [List 1.5] と同じ意味のプログラムは C++ では次のように書くことができる.

[List 1.6]

```
 1:  #include <iostream>
 2:
 3:  void inc(int& x) {  // 参照渡しを宣言
 4:    x++;  // x に * などを付ける必要は無い
 5:    std::cout << "x = " << x << std::endl;
 6:  }
 7:
 8:  int main(void) {
 9:    int a = 5;
10:    inc(a);  // 呼び出し時にも & は不要
11:    std::cout << "a = " << a << std::endl;
12:    return 0;
13:  }
```

この実行の結果は,

```
x = 6
a = 6
```

となる.

C++ のこの構文の方が簡便であるが, プログラムの可読性の観点から, 引数の値が変更される可能性が関数名などから明らかでない場合は, 従来通りポインタを用いた `inc(&x)` の構文を用いるか, `x=inc(x)` と返り値を代入するなど, 明示的に値が変ることを示す方がよいとされているので, 乱用には注意.

課題 1.2 2 つの整数型の変数 x と y を受け取り, x を y で割った商を x に, 剰余を y にセットする関数 `divrem(int&, int&)` を作成し, 実行結果を確認せよ.

1.3.2 参照の返り値

- 引数だけでなく, 関数の返り値も参照にすることができる. 実質的には C 言語のポインタを返す関数と同じと考えることができる. その用い方については, 後のセクションで述べる.

1.3.3 関数のオーバロード

- オーバロードとは

  C 言語では, 同じ名前の関数は唯一つ存在が許されている. これに対し C++ では, 同じ名前でも引数の数とそれぞれの型が異れば, それは異なる関数と見なされる.

  次の例では, add 関数が 3 回定義されているが, 引数の数や型が違うので, 異なる 3 つの関数として扱われる.

[List 1.7]

```
 1:  #include <iostream>
 2:
 3:  int add(int a, int b) {return a+b;}           // (1)
 4:  int add(int a, int b, int c) {return a+b+c;}  // (2)
 5:  double add(double a, double b) {return a+b;}  // (3)
 6:
 7:  int main(void) {
 8:    int x=3, y=4, z=5;
 9:    double p=3.14, q=2.44;
10:    std::cout << add(x,y) << std::endl;    // (1) が呼ばれる
11:    std::cout << add(x,y,z) << std::endl;  // (2) が呼ばれる
12:    std::cout << add(p,q) << std::endl;    // (3) が呼ばれる
13:    return 0;
14:  }
```

- 10˜12 行目の呼出しに対しては, コンパイラが引数の型を調べ, 型が一致した add が選択され呼び出される. 型が一致するものがなかったり, 複数の型の解釈が考えられる場合にはコンパイルエラーとなる.
- このため, C++ では, 関数名だけでなく, add(int,int), add(int,int,int), add(double,double) のように, 関数名と引数の型のリストを組にして関数を識別するのが習慣になっている.
- C 言語ではエラーとなり, 気のきいたコンパラだと「add が 2 回宣言されている」とか「add の引数の数が合っていない」などのエラーメッセージが出される (はず).

このような仕組みを「一つの名前に多くの役割を負わせる」という意味で**オーバーロード (overloading)** と呼ぶ. 同じ意味を持った関数でも, 引数の数や型によって処理が違ってくることがあるので, その場合の名前付けに便利である.

1.3.4 関数のインライン展開

- インライン展開とその必要性
  - 次のような簡単な関数 max(int,int) (2 つの整数引数のうち, 大きい方を返す) を考える.

[List 1.8]

```
1:  #include <iostream>
2:
3:  int max(int a, int b) {return a>b ?  a :  b;}
4:
5:  int main(void) {
6:    int x, y, z;
:    ...
7:    z = max(x+1,y) + 3;
:    ...
8:    return 0;
9:  }
```

  - インライン展開とは, main(void) 内の max(int,int) に対して関数呼び出しのコードを生成するのではなく, 下のように max(int,int) の内容を main(void) に展開して関数呼び出しの処理を省略するというもの.

(人手による変換ではなく, コンパイラが自動的に行ってくれることを意味する.)

[List 1.9]

```
1:  int main(void) {
2:    int x, y, z;
:    ...
4:    z = ((x+1>y) ?  x+1 :  y) + 3;
:    ...
5:    return 0;
6:  }
```

– プログラムの可読性や, 後に述べるクラスの定義のためには, 1~ 数命令で済む操作でも関数として定義する方が好ましいことが多い. しかし, 関数呼び出しを行うためには, レジスタの退避, パラメータの設定, フレームの作成, 分岐, 返り値の設定, レジスタの回復などを行わなければならないので, 簡単な関数に対していちいち関数呼び出しの処理を行うと効率が悪い. そこで, インライン展開をコンパイラに指示することにより, その両立を図るわけである.

- インライン展開の指示
  – 関数の冒頭にキーワード `inline` を追加すれば良い.

[List 1.10]

```
3':  inline int max(int a, int b) {return a>b ?  a :  b;}
```

  – `inline` 宣言すれば必ずインライン展開が保証されるわけではない. まず, 関数の定義が同じファイルにあることが必要である. また, 実際にインライン展開をして得をするかどうかの判断にも依存する. これは当然ながらコンパイラの賢さによって変わって来る.
  – 基本的に `#define` で引数つきのマクロを定義するのと同じ効果が得られるが, 種々の点で `inline` 宣言付き関数の方が優れていると言われている. 例えば, 良く知られる,

```
#define max(a,b) ((a)>(b)?(a):(b))
```

という定義に対して, インクリメント演算子を用いると,

```
a = 0;
max(++a,0);   // a のインクリメントは 2 回行われる?
max(++a,10);  // a のインクリメントは 1 回だけ行われる?
```

という妙なことが起こってしまう[1]. が, インライン関数を用いればこのような問題は生じない.

1.4 クラス—例 (1)「スタック」

- クラス (class) とは
  – いわゆる「オブジェクト指向プログラミング」の中心的な概念で, C++ にとっても最も重要な概念の一つである.
  – 最も単純に言えば「クラス＝構造体＋そのデータを操作する関数群」で, 一つの「新しいデータ型の定義」と考えることもできる.

[1] `max` を展開してみよ. `( ++a > x ? ++a : x )` において, `++a` が `x` より大きいと `++a` は 2 回, 小さいと 1 回評価される.

1.4.1 「整数型スタック」クラスの例

- まず,「整数型のスタック」というクラスの定義例を示す.
  - クラスの仕様の定義
  - メンバ関数の詳細
  - 利用例

  のコードを示す.
- クラスの仕様定義
  - C 言語の struct の定義と大体同じで, スタックを構成するデータ項目を宣言しているが, その他にスタックを操作するための関数が定義されている点が大きな違いである.

[List 1.11]

```
 1:  #include <iostream>
 2:  #include <assert.h> // assert を使うためのヘッダ
 3:
 4:  class stack {
 5:    private:
 6:      static const int max = 100;
 7:      int data[max];
 8:      int sp;
 9:    public:
10:      void push(int);      // データをプッシュする
11:      void pop();          // データをポップする
12:      bool empty() const; // スタックが空のとき true を返す
13:      int top() const;     // スタックトップのデータを返す
14:      int size() const;    // 現在格納されているデータ数を返す
15:      stack();             // コンストラクタ
16:      ~stack();            // デストラクタ
17:  };
```

  - 一番最後 (17 行目) のセミコロン (;) は省略不可. 忘れないように! (忘れると, 山のようにエラーメッセージが出る.)
  - 6˜ 8: スタックを構成するデータを定義している. max は配列のサイズ, data[max] は整数配列, sp はスタックポインタ. これらのデータはこの stack の**メンバ (member)** と呼ばれる.
  - 10˜ 14: スタックを操作する関数の宣言をしている. これらの関数は**メンバ (member)** 関数あるいはメソッド (method) と呼ばれる.

push(int d)	整数値 d をスタックにプッシュする
pop()	スタックをポップする
empty()	スタックが空のとき 1, そうでないとき 0 を返す
top()	スタックトップの値を返す
size()	スタックに入っているデータ数を返す

  - empty(), top(), size() には const が付記されているが, これは, その関数によってそのクラスのメンバーの値が更新されないことを表す. (const は付記しなくても当面は問題ないが, 型チェックでコンパイルエラーとなることがある.) push(int) や pop() では, メンバーの値が更新されるので const を付記してはならない (コンパイルエラーになる).

– 15˜ 16: stack() と ∼stack() もメンバ関数であるが, 特別な意味を持っている. stack() は stack を生成したときの初期化を行う関数で**コンストラクタ (constructor; 構築子)** と呼ばれる. ∼stack(); はその逆に stack が消去させるときに呼ばれる関数で**デストラクタ (destructor; 解体子)** と呼ばれる.

– 5: private は, それ以下 (6˜ 8 行目) のデータが外部から直接にはアクセスできないことを宣言している.

– 9: public は, 逆にそれ以下 (10˜ 16 行目) のメンバ関数が外部から使用であることを宣言している.

☆ すなわち, この stack は, それを実現しているデータを直接操作することはできず, public と宣言されたメンバ関数を通じてのみ操作できるようになっている. こうしておけば, stack の内部のデータ構造や操作アルゴリズムを変更する必要が生じても (例えば配列でなく線形リストで書き直すなど), どの部分を書き直せばよいか分かりやすいし, 変更の影響が stack を利用しているプログラムには全く及ばない. このような概念は「データのカプセル化 (encupslation of data)」と呼ばれ, オブジェクト指向プログラミングのメリットの一つと言われている.

– 2: <assert.h> は, 後に assert(bool) 関数を用いるので, そのためのヘッダを宣言している.

- メンバ関数の実体

  7 つのメンバ関数の実体は次のようになる.

[List 1.12]

```
 1:  void stack::push(int d) {
 2:      assert(sp<max);
 3:      data[sp++] = d;
 4:  }
 5:
 6:  void stack::pop() {
 7:      assert(0<sp);
 8:      --sp;
 9:  }
10:
11:  bool stack::empty() const {
12:      return sp==0;
13:  }
14:
15:  int stack::top() const {
16:      return data[sp-1];
17:  }
18:
19:  int stack::size() const {
20:      return sp;
21:  }
22:
23:  stack::stack() {sp = 0;}
24:
25:  stack::∼stack() {}
```

– 関数の先頭の stack:: は, この関数が stack クラスのメンバ関数であることを示している.

– 2: assert(sp<max); は, スタックオーバフローの検査のため, sp<max が成立しているかどうかをチェックしている. assert (条件式); は, 条件式の値が 0 になるとプログラムを強制終了し, その行番号を

表示する.

- 7: 同様にスタックアンダフローの検査.
- 11, 15, 19: 宣言で const を指定した場合は, 実装にも const を指定すること.
- 23: コンストラクタは, スタックの初期化として, スタックポインタを 0 にしている.
- 25: デストラクタでは, 特にすることはない.

- 利用例 (1)

  こうして定義したスタックは, 以下のコードで利用できる.

[List 1.13]

```
 1:  int main(void) {
 2:      stack s;  // stack 型変数の宣言
 3:      s.push(5);  // s = (5)
 4:      s.push(8);  // s = (5 8)
 5:      s.push(9);  // s = (5 8 9)
 6:      std::cout << s.top() << std::endl;  // 9 を表示
 7:      s.pop();  // s = (5 8)
 8:      std::cout << s.top() << std::endl;  // 8 を表示
 9:      s.pop();  // s = (5)
10:      s.push(3);  // s = (5 3)
11:      std::cout << s.size() << std::endl;  // s のサイズは 2
12:      while(!s.empty()) {
13:        std::cout << s.top() << std::endl;
14:        s.pop();
15:      }
16:      return 0;
17:  }
```

- 2: スタックの宣言. 構文的には int i; と宣言するのと同じ. s という stack 型の変数が一つ生成されるが, この際にコンストラクタが呼ばれる.
- s に対するメンバ関数 push(5) の呼び出しは, s.push(5) のように構造体のデータに対するアクセスと同じ構文で行う. (ちなみに, p がスタックに対するポインタの場合は, p->push(5) という書き方が許される.)
- 変数 s は main(void) 関数の終了時に消滅する. この際にデストラクタが呼ばれる.

課題 1.3 スタックのプログラムを入力し ([List 1.11], [List 1.12], [List 1.13] をこの順に一つのファイルに打ち込む), コンパイルし実行してみよ. 実験状況・感想とともに, 作成したプログラムをレポートに添付すること.

- 利用例 (2)

  int などと同様の構文で, stack の変数も動的に割り当てたり解放したりすることができる.

[List 1.14]

```
 1:  int main(void) {
 2:      stack* s = new stack;  // スタックの動的割当て
 3:      s->push(5);  // メンバ関数の呼び出し (ポインタバージョン)
 4:      s->push(8);
 5:      s->push(9);
 6:      std::cout << s->top() << std::endl;
 7:      s->pop();
 8:      std::cout << s->top() << std::endl;
 9:      s->pop();
10:      s->push(3);
11:      std::cout << s->size() << std::endl;
12:      while(!s->empty()) {
13:        std::cout << s->top() << std::endl;
14:        s->pop();
15:      }
16:      delete s;  // スタックの解放
17:      return 0;
18:  }
```

- 2: stack を指すポインタ変数 s を宣言し, new で割り当てた新しいスタックへのポインタを代入している. この割当ての際 (正確には割当ての直後) にコンストラクタが呼ばれる.
- 16: スタック s の領域を解放している. この際 (正確にはこの解放の直前) にデストラクタが呼ばれる.

☆ stack の配列を作ることも可能である.

1.4.2 メンバ関数のインライン化

- stack のメンバー関数 は, インライン宣言して呼び出しオーバヘッドを無すことができる.
- 関数の宣言の頭にキーワード inline を追加しても良いが, メンバ関数の場合は, クラスの宣言中に直接本体を書けば, インライン宣言したことになる. 具体的には, 次のように書く.

[List 1.15]

```
 1:  class stack {
 2:      private:
 3:        static const int max = 100;
 4:        int data[max];
 5:        int sp;
 6:      public:
 7:        void push(int d) {assert(sp<max); data[sp++] = d;}
 8:        void pop() {assert(0<sp); --sp;}
 9:        bool empty() const {return sp==0;}
10:        int top() const {return data[sp-1];}
11:        int size() const {return sp;}
12:        stack() {sp = 0;}
13:        ~stack() {}
14:  };
```

☆ あまり長いものはインライン指定しないこと．コンパイルに時間がかかったり，オブジェクトコードが大きくなったりすることがある．クラス宣言の中に書くのは，1 行で書ける程度が目安と言われている．

1.4.3 内部データの動的割当て

- スタックの配列などの内部データは，動的に割り当てても良い．(これによりスタックのサイズを可変にするなど，可変長のデータが扱える．)
- スタックサイズの可変化
  - stack の宣言時に，stack s(n); と書けば，内部配列としてサイズ n のものが確保されるようにする．
  - 配列 data を動的に割当てる．これはコンストラクタで行う．サイズ n はコンストラクタの引数として指定するようにする．
  - スタック s が不要になった時点で，動的に割当てた配列 data の領域を解放しなければならない．この解放はデストラクタで行う．
  - [List 1.15] の 12 行目を次のように変更し，受け取ったサイズ (sz) の配列を動的に割り当てるようにする．

    [List 1.16]

    ```
    12.1:  stack(int sz=100) {
    12.2:    sp = 0;
    12.3:    max = sz;
    12.4:    data = new int[max];
    12.5:  }
    ```

  - 12.1: コンストラクタのパラメータに int sz=100 と値が代入されているが，これは**デフォルト引数**と呼ばれるもので，引数が指定されなかった場合にはデフォルト値として 100 が代入されるというものである．
  - new で割り当てた領域は，delete で解放しなければならない．この解放はデストラクタで行う．[List 1.15] の 13 行目を

    [List 1.17]

    ```
    13':  ~stack() {delete [] data;}
    ```

    とする．
  - [List 1.13] の 2 行目で，例えばサイズ 45 を指定してそのサイズのスタックを割り当てる．

    [List 1.18]

    ```
    2':  stack s(45);
    ```

  - 引数を指定しなかった場合には，デフォルト値が指定されたものとして処理される．即ち，

    ```
    stack s;
    ```

    と

    ```
    stack s(100);
    ```

    は等価になる．

課題 1.4 前の演習で作成したスタックのプログラムを, 配列を動的に割当てるバージョンに変更せよ. コンストラクタ中で max の値を表示するなどして, 指定したサイズの配列が割当てられていることを確認せよ.

※ 上記リストの 12.1 行目のように, max に値を代入する場合, max の宣言が static const int (定数の int) のままではではまずい (定数ではなくなるので).

※ 当然だが, data の宣言が int data[max] のままなのもまずい. data をポインタとして宣言すること.

1.4.4 フレンド関数

クラスに関連した関数の扱いについて少し解説する.

- 2 つの stack の内容が等しいかどうか判定する関数 equal(const stack&, const stack&) を作ってみる.

  ※ 引数の const は, この関数の内部でその引数が書き換えられないことを表す.

  – equal(const stack&, const stack&) は 2 つのスタックのサイズが等しく, かつ, データが全て一致しているとき true を返し, そうでないとき false を返す.

  (注) ここでは true と false という 2 つの値をとる bool という型を使う.

  ```
  typedef int bool;
  const bool false = 0;
  const bool true = 1;
  ```

  と定義したのと大体同じと考えて良い.

  – スタックのような大きなデータは, 値引数で渡すとコピーが作られることになって効率が悪いので, 値渡し (stack) ではなく参照渡し (stack&) を用いるようにする. (関数中でデータの書き換えが起ると困る場合は除く.)

  – 関数呼び出しによって, 参照渡しされた stack が書き換えられることが無い場合 (すなわち read-only の場合) には, const を宣言する. (宣言しなくてもエラーにはならないが, ある状況の組合せで警告が出ることがある.)

- アルゴリズムは簡単で, 関数は例えば次のように作れる.

[List 1.19]

```
 1:  bool equal(const stack& s1, const stack& s2) {
 2:    bool eq = true;  // 等しければ eq=true, そうでなければ eq=false
 3:    if (s1.sp!=s2.sp) eq = false;  // データ数が違えば等しくない
 4:    else {
 5:      for (int j=0; j<s1.sp && eq; j++) {
 6:        if (s1.data[j]!=s2.data[j]) eq = false;
 7:        // データが一つでも一致しなければ等しくない
 8:      }
 9:    }
10:    return eq;
11:  }
```

- 使い方は次の通り. 10 行目では 0 が, 12 行目では 1 が出力されるはず.

[List 1.20]

```
 1:  int main(void) {
 2:    stack s1, s2;
 3:    s1.push(5);
 4:    s1.push(8);
 5:    s1.push(9);
 6:    s2.push(5);
 7:    s2.push(8);
 8:    s2.push(9);
 9:    s2.push(10);
10:    std::cout << equal(s1,s2) << std::endl;
11:    s2.pop();
12:    std::cout << equal(s1,s2) << std::endl;
13:    return 0;
14:  }
```

- しかし, これをコンパイルしてみると, エラーが出る. (コンパイラによりメッセージは異なるが.)

```
sstack2.cpp:  In function ‘bool equal(const class stack &, const class stack &)’:
sstack2.cpp:30:  member ‘sp’ is a private member of class ‘stack’
sstack2.cpp:30:  member ‘sp’ is a private member of class ‘stack’
sstack2.cpp:32:  member ‘sp’ is a private member of class ‘stack’
sstack2.cpp:33:  member ‘data’ is a private member of class ‘stack’
sstack2.cpp:33:  member ‘data’ is a private member of class ‘stack
```

  - これは, equal(const stack&, const stack&) は stack のメンバー関数ではないので, private 宣言された内部データにはアクセスできない, ということを意味している.
  - メンバー関数は, ある一つのクラスのデータに対してその操作を行うものである. equal(const stack&, const stack&) は, 2 つのスタックを操作するものなので, メンバー関数として宣言することはできない.
  - 内部データを public 宣言すれば, コンパイルは可能になるが, 他のプログラムからも直接スタックの内部を操作できることになり, データのカプセル化が損なわれてしまう.
  - この equal(const stack&, const stack&) のように, メンバー関数ではないがクラスの内部データを操作する必要がある, という関数はを作らなければならない場合が生じる. このような場合には, そのような関数に対し, クラス定義の中で内部データへのアクセスを許可する宣言を行うことができる. きる. それが「フレンド」宣言であり, このような関数を「フレンド関数」と呼ぶ.

    具体的には, 12 行目のようにキーワード friend をつけて関数の宣言を追加すれば良い.

    [List 1.21]

```
 1:  class stack {
 2:    private:
 3:      static const int max = 100;
 4:      int data[max];
 5:      int sp;
```

```
 6:    public:
 7:      void push(int);
 8:      void pop();
 9:      bool empty() const;
10:      int top() const;
11:      int size() const;
12:      friend bool equal(const stack&, const stack&);
13:      stack();
14:      ~stack();
15:  };
```

課題 1.5 stack に bool sum_equal(const stack&, const stack&) を追加したバージョンを作成し, 動作を確認せよ. この関数は, 2 つのスタックに蓄えられている数の和が等しい時 true を, そうでないとき false を返す.

☆ フレンド関数は, 便利ではあるが, せっかくのデータ隠蔽をあっさり破壊してしまうため, 乱用は避けるべき (本当に必要なケースはほとんどない) とされている.

1.5 クラス—例 (2)「複素数」

- 2 つ目の例として複素数のクラス Complex を定義してみる.

  保持すべきデータは, 実部と虚部を表す二つの浮動小数点数 real, imag である. 関数としては, 値の設定や実部・虚部の参照, データの書き出しの他に, 加減算, 乗算を定義する.

- クラスの宣言部は次の通り.

[List 1.22]

```
 1:  class Complex {
 2:    private:
 3:      double real;  // 実部
 4:      double imag;  // 虚部
 5:    public:
 6:      Complex() {real = 0.0; imag = 0.0;}
 7:      Complex(double r, double i) {real = r; imag = i;}
 8:      ~Complex() {}
 9:      double re() const {return real;}  // 実部を取り出す
10:      double im() const {return imag;}  // 虚部を取り出す
11:      void set_re(double r) {real = r;}  // 実部を設定
12:      void set_im(double i) {imag = i;}  // 虚部を設定
13:      void print(std::ostream& os) const {  // 出力
14:        os << real << "+" << imag << "i";
15:      }
16:  };
```

- 例えば, 次のような使い方ができる. 変数 a, b, c に異なる方法で値の設定を行い (詳細は後の解説参照), その値を出力している.

[List 1.23]

```
 1:  int main(void) {
 2:    Complex a;  // 値の設定の仕方その 1
 3:    a.set_re(1.11);
 4:    a.set_im(2.22);
 5:    Complex b(3.33,4.44);  // 値の設定の仕方その 2
 6:    Complex c;  // 値の設定の仕方その 3
 7:    c = Complex(9.99, 88.88);
 8:    a.print(std::cout); std::cout << std::endl;
 9:    b.print(std::cout); std::cout << std::endl;
10:    c.print(std::cout); std::cout << std::endl;
11:    c = a;  // 普通の代入も行える
12:    c.print(std::cout); std::cout << std::endl;
13:    return 0;
14:  }
```

- 2 行目で, 引数のないコンストラクタ Complex() により Complex 型の変数を作り (この時点での値は $0+0i$), 3˜ 4 行目で値を設定している.
- 5 行目では, 2 つの引数を持つコンストラクタ Complex(double, double) により, 変数を作ると同時に値を設定している.
- 6 行目では, 2 行目と同様に変数を作り, 7 行目では名前のない Complex 型データをつくってそれを代入している.

課題 1.6 上の [List 1.22], [List 1.23] をコンパイルし, 結果を確認せよ. その後, double 型の引数 r をひとつだけとって, 実部に r の値を設定し虚部には 0 を設定するようなコンストラクタ Complex(double) を作成し, main(void) にそれを使うコードを追加せよ.

- Complex 型の加算を定義する.
  - C 言語的な方法としては, add(Complex, Complex) を

    [List 1.24]

    ```
    1:  Complex add(Complex a, Complex b) {
    2:    double r = a.re() + b.re();
    3:    double i = a.im() + b.im();
    4:    return Complex(r,i);
    5:  }
    ```

    のように定義すれば,

[List 1.25]

```
 1:  int main(void) {
 2:    Complex x, y, z, a;
 3:    x = Complex(1.00, 2.00);
 4:    y = Complex(2.22, 3.14);
 5:    z = Complex(4.23, 9.99);
 6:    a = add(add(x,y),z);
 7:    x.print(std::cout); std::cout << std::endl;
 8:    y.print(std::cout); std::cout << std::endl;
 9:    z.print(std::cout); std::cout << std::endl;
10:    a.print(std::cout); std::cout << std::endl;
11:    return 0;
12:  }
```

のように加算を行うことができる.

– C++ ではさらにカッコよくて便利な演算定義/呼出し方がある. 関数名に英数字からなる識別子だけでなく, + や - などの演算子を用いることができ, それを式の中に用いることができる. これを用いると,

```
a = add(add(x,y),z);
```

は,

```
a = x + y + z;
```

と書くことができるようになる. さらに, * や / を用いた場合には, 括弧を用いなくても自動的に+ や - より優先して実行してくれる. つまり, 自分の作ったクラスに対して, +, -, *, / などの演算が自由に定義できるわけである.

その定義には次のような構文を用いる.

[List 1.26]

```
1:  Complex operator+(Complex a, Complex b) {
2:    double r = a.re() + b.re();
3:    double i = a.im() + b.im();
4:    return Complex(r,i);
5:  }
```

* [List 1.24] の 1 行目の関数名「add」を「operator+」 に変えるだけである.
* 「operator+」 は 「+ という演算子」を意味すると考えればよい.
* 「operator+」 の引数は必ず 2 個である.
* この「operator+」の定義は, int 型や double 型に対して始めから定義されている「operator+」のオーバロードであると考えられる. (関数名は同じだが, 引数の型が違うので違う関数が呼び出される.)
* 文頭にキーワード inline を付加すれば, 関数呼び出しでなくインライン展開により処理させることができる.

課題 1.7 double と Complex の乗算, および Complex 同士の乗算を定義せよ. 次のようなメインルーチンで実行結果を確かめよ.

[List 1.27]

```
 1:  int main(void) {
 2:    Complex a(1.00, 2.00);
 3:    Complex b(3.00, 4.00);
 4:    Complex c(2.35, 5.32);
 5:    Complex x = 2.0 * c;
 6:    x.print(std::cout); std::cout << std::endl;
 7:    Complex y = a * b;
 8:    y.print(std::cout); std::cout << std::endl;
 9:    return 0;
10:  }
```

実行して

```
4.7+10.64i
-5+10i
```

が出力されることを確認せよ.

- Complex クラスのデータを, << を用いて出力できるようにする.
  - 出力の << 演算は, + などと同様に一つの演算子であり, operator<< で表せる.

[List 1.28]

```
1:  std::ostream& operator<<(std::ostream& os, Complex& c) {
2:    c.print(os);
3:    return os;
4:  }
```

  - 関数を void 型ではなく std::ostream& 型とし, 3 行目で受取った os を返すのは, 連続出力 (std::cout << "a = " << a << std::endl など) を行うため. std::ostream を値として返すと std::ostream 型の変数のコピーが作られてしまうので, その参照を返すようにしている (この辺りは少し難しいので, 完全に理解できなくても良い).
- Complex クラスの配列, ポインタ, 参照, 動的配列
  - 組込みの int 型や double 型と全く同様に行える.

[List 1.29]

```
 1:  int main(void) {
 2:    Complex a[3];  // a[0], a[1], a[2] ができる
 3:    a[0] = Complex(1.00,2.00);
 4:    a[1] = a[0] + Complex(0.00,1.00);
 5:    Complex *c = &(a[2]);  // c は a[2] を指すポインタ
 6:    *c = a[0];  // *c の実体は a[2] となる
 7:    for (int i=0; i<3; i++) std::cout << a[i] << std::endl;
 8:    Complex *b = new Complex[3];  // 動的な配列の割当て
 9:    for (int i=0; i<3; i++) b[i] = a[i] + Complex(1.00,1.00);
10:    for (int i=0; i<3; i++) std::cout << b[i] << std::endl;
11:    delete [] b;  // b の解放
12:    return 0;
13:  }
```

課題 1.8 上の [List 1.29] を実行してその結果を示せ. (プログラムを提出せよ.)

参考図書

1. Bjarne Stroustrup 著, (株) ロングテール/長尾高弘 訳, “プログラミング言語 C++ [第 3 版],” アジソンウェスレイ/アスキー, ISBN4-7561-1895-X, 7,000 円.

   最近の言語拡張や STL まで含んだ完全なマニュアル. 高いので個人で買うのは辛いが, 研究室に入ったら研究室で 1 冊買ってもらおう.

2. Herbert Schildt 著, 神林靖 監修, トップスタジオ 訳, “独習 C++ 改訂版,” 翔泳社, ISBN4-88135-761-8, 3,200 円.

   一から C++ を勉強していこうという人に.

3. Gregory Satir, Doug Brown 共著, 望月康司 監訳, 谷口功 訳, “C++ プログラミング入門” オライリー/オーム社, ISBN4-900900-04-4, 3,900 円.

   C を一通り理解している人が C++ を詳しく勉強するのに.

4. Scott Meyers 著, 吉川邦夫 訳, “Effective C++ [改訂 2 版],” アジソンウェスレイ/アスキー, ISBN4-7561-1808-9, 3,800 円.

   お勧めの一冊. C を理解していることが前提になるが, C++ のエッセンスや慣習等「常識」が詳しく勉強できる.

5. Scott Meyers 著, 安村通晃, 伊賀総一郎, 飯田朱美 訳, “More Effective C++,” アジソンウェスレイ/アスキー, ISBN4-7561-1853-4, 3,800 円.

   Effective C++ の続編. こちらはかなり難しい内容. (訳が悪い.)

6. Andrew Koenig, Barbara E. Moo 共著, 小林健一郎 訳, “C++ 再考,” アジソンウェスレイ/星雲社, ISBN4-7952-9721-5, 3,800 円.

   これが読破できれば一応 C++ は極められた, というオタク向けの本. 3. の内容程度は理解していないと読めない.

Nagisa ISHIURA