情報システム工学演習 II
C++プログラミング 第2回

谷口 一徹

2 標準ライブラリ

ここでは, C++ の標準ライブラリとその使い方の概要を紹介する.

2.1 string クラス

2.2 STL コンテナクラス

2.3 共通アルゴリズム

2.1 string クラス

- 特長
  - C 言語では, 文字列は ”\0 ” で終る文字の配列として表現したが,
    - 記憶制御 (配列の割当て, 解放) が面倒
    - 基本操作 (コピー, 比較等) が関数呼び出しになり, 書くのが面倒/読みにくい
    - 連結, 書き換え等, 長さが変わる操作のコーディングが大変複雑になる,

    等の問題があった.
  - C++ では, 文字列のクラスとして様々なものが提案され, その標準化には紆余曲折があったが, 現時点では string クラスの使用が推奨されている.
  - string クラスでは, 文字列の内部表現がカプセル化されており, 記憶制御の煩わしさが封じ込められているとともに, オペレータの使用により文字列の演算が書き易くなっている.
- 宣言, 初期設定, 代入, 入出力
  - `<string>` を include する.
  - 宣言, 初期設定, 代入は, ほぼ int 等の組込みのデータ型と同じように行える.
  - 可変長の文字列が扱える. 記憶領域の制御は自動的に行われるので, 長さや現在割り当てられている領域のサイズを意識する必要は無い.
  - 出力は<<で行える.
  - 入力は>>で行える.
    - >> による入力では, 空白, 改行, タブなどの空白文字で区切られた「単語」が入力される.
    - 空白などに区切られて困る場合は, getline(入力ストリーム, string 変数) を使えば, 一行を変数に読み込むことが出来る.

```
[List 2.1]
 1: #include <iostream>
 2: #include <string>
 3:
 4: int main()
 5: {
 6:   std::string message;
 7:   std::string s;
 8:   message = "Enter strings";
 9:   std::cout << message << ": ";
10:   std::cin >> s;
11:   std::cout << "word 1 = " << s << std::endl;
12:   std::cin >> s;
13:   std::cout << "word 2 = " << s << std::endl;
14:   getline(std::cin,s);
15:   std::cout << "line = " << "'" << s << "'" << std::endl;
16:   getline(std::cin,s);
17:   std::cout << "line = " << "'" << s << "'" << std::endl;
18:   return 0;
19: }
```

Enter strings: というプロンプトに対して

```
This is a pen.
```

と入力すると,

```
word1 = This
word2 = is
line = ' a pen.'
```

と出力される. 更に 1 行

```
That is a book.
```

と入力すると,

```
line = 'That is a book.'
```

が出力される.

- 連接, 比較
  - 連接は+ により行える. += 演算も使える. s+=a で s の末尾に a のコピーが連接される.
  - == は文字列が等しいかどうかの比較. <, <=, >, >= は文字列の辞書順の比較.

```
[List 2.2]
 1: #include <iostream>
 2: #include <string>
 3:
 4: int main()
 5: {
 6:   std::string a = "Osaka ";
```

```
 7:   std::string b = "Prefectural ";
 8:   std::string c = "University";
 9:   std::string p, q, r;
10:   p = a + c;
11:   q = a + b + c;
12:   r = a;
13:   r += c;
14:   std::cout << "p = " << p << std::endl;
15:   std::cout << "r = " << q << std::endl;
16:   std::cout << "r = " << r << std::endl;
17:   if (p==q) std::cout << "p==q" << std::endl;
18:   else if (p<q) std::cout << "p<q" << std::endl;
19:   else std::cout << "p>q" << std::endl;
20:   if (p==r) std::cout << "p==r" << std::endl;
21:   else if (p<r) std::cout << "p<r" << std::endl;
22:   else std::cout << "p>r" << std::endl;
23:   return 0;
24: }
```

これを実行すると,

```
p = Osaka University
r = Osaka Prefectural University
r = Osaka University
p>q
p==r
```

が出力される.

- 文字列長, 部分文字列, 置換
  string 型データ s に対して, 次のような関数が用意されている

  1. s.length() … s の長さ (文字数)
  2. s[i] … s の i 番目の文字 (先頭は s[0]). 代入も可能. s = ”woman” に対して s[3]= ’ e ’ とすれば, s は”women” になる.
  3. s.substr(i,k) … s の i 番目以降 k 文字を抽出して得られる部分文字列. s = ”Osaka Univ.” に対して, s.substr(4,6) は”a Univ” になる. 2 番目の引数 k を省略すると, i 番目から最後までを返す. s = ”Osaka Univ. ” に対して, s.substr(6) は”Univ.” になる.
  4. s.find(t) …s 中に文字列 t が含まれるかどうかを先頭から探し, 見つかればその先頭文字の位置を返す. 見つからなければ, 無効な文字位置を表す s.npos を返す. 例えば, s = ”to be or not to be” のとき, s.find(”be”)==3 となり, s.find(”and”)==s.npos となる.
  5. s.replace(i,k,t) … s の i 番目以降の k 文字を, 文字列 t で置換する. t の長さが k である必要は無く, s の長さは t の長さに応じて延び縮みする. 例えば, s = ”I live in Osaka” に対して, s.replace(2,7,”love”) とすると s は”I love Osaka” になる. 文字列 s 中にある”often” を”usually” に置き換えるには, s.replace(s.find(”often”),5,”usually”) とすればよい. (s 中に必ず ” often ” が含まれている場合.)
  6. s.insert(i,t) … s の i 番目の文字の前に文字列 t を挿入する.

【EX-2.1】次のプログラムは, 名前を入力するとその電話番号を出力するプログラムである. 名前と電話番号は Entry というクラスの配列で記憶している. Entry というクラスは, string name に名前を, string phone に電話番号を格納するものである. このプログラムは,

- 大阪の電話番号の局番を 4 桁化する. (古い話ですみません.“06-xxx-xxxx”を“06-6xxx-xxxx”に書き換える, という意味です.)
- 入力した文字列に一致する名前と電話番号を出力する. (厳密に一致するものだけ出力しても良いし, 入力した文字列を部分文字列として含むものを全て出力しても良い).

というものである. このプログラムの空白を適当に埋めて完成させよ.

[List 2.3]

```
 1: #include <iostream>
 2: #include <string>
 3:
 4: class Entry {
 5:   public:
 6:     std::string name;
 7:     std::string phone;
 8:     Entry(const std::string& nm="", const std::string& ph="") {
 9:       name = nm;
10:       phone = ph;
11:     }
12: };
13:
14: std::ostream& operator<<(std::ostream& os, const Entry& e) {
15:   os << e.name << ": " << e.phone;
16:   return os;
17: }
18:
19: int main()
20: {
21:   Entry e[10];
22:   int n = 0;
23:   e[n++] = Entry("university of tokyo", "03-3812-2111");
24:   e[n++] = Entry("osaka university", "06-879-7806");
25:   e[n++] = Entry("kinki university", "06-721-2332");
26:   e[n++] = Entry("osaka prefectural university", "0722-52-1161");
27:   e[n++] = Entry("kyoto university", "075-753-7531");
28:   for (int i=0; i<n; i++) {
        *** 大阪の局番 4 桁化 ***
29:   }
30:   std::cout << "directory service: ";
      *** 文字列を入力し, その電話番号を検索し, 出力 ***
31:   return 0;
32: }
```

2.2 STL コンテナクラス

- STL とは
  - Standard Template Library の略で, C++ で用いられるデータ構造やアルゴリズムの標準的なライブラリである.
  - リスト, 可変配列, 連想配列, 集合などのコンテナ (container; データを保持するデータ構造) と, その上で動作する探索, ソーティングなどのアルゴリズムが提供されている.
  - これらのコンテナは, int, string などの基本的なデータ型だけでなく, あらゆるクラスを保持することができ, アルゴリズムもこれらを扱うことが出来る. ユーザが定義した型の変数のリストや, リストの集合等も扱うことができる.
  - STL の利用により, リスト処理等の動的記憶管理や, 二分木処理, 探索, ソーティングなどのコーディングを行う煩わしさから解放される.

2.2.1 vector

- vector は可変配列である.
  - 必要に応じてサイズが変更される.
  - 末尾にデータを追加したり削除するスタック演算が用意されている.
- 宣言と基本操作
  - <vector> を include する
  - 型 T のデータを保持する vector は, vector<T> と書く. 例えば, 10 個の要素を整数を保持する配列は, 普通の配列では int a[10] と宣言するところを, vector を用いる場合には, vector<int> a(10); と宣言する.
  - 通常の配列と同様, i 番目の要素は a[i] で読み書きできる.
  - 配列サイズは, メンバ関数 size() で参照できる.
  - 配列全体の代入 (コピー) も行える.

[List 2.4]

```
 1: #include <iostream>
 2: #include <vector>
 3:
 4: int main() {
 5:   std::vector<int> a(5);
 6:   std::cout << "a.size = " << a.size() << std::endl; // a.size=5 と表
示
 7:   for (int i=0; i<a.size(); i++) a[i] = i;
 8:   for (int i=0; i<a.size(); i++) std::cout << a[i] << " ";
 9:   std::cout << std::endl; // 0 1 2 3 4 と表示
10:
11:   std::vector<int> b(3);
12:   b[0] = 7; b[1] = 5; b[2] = 3;
13:
14:   std::vector<int> c(10);
```

```
15:   for (int i=0; i<c.size(); i++) c[i] = i*i;
16:
17:   a = b; // b が a にコピーされる
18:   std::cout << "a.size = " << a.size() << std::endl; // a.size=3 と表
示
19:   for (int i=0; i<a.size(); i++) std::cout << a[i] << " ";
20:   std::cout << std::endl; // 7 5 3 と表示
21:
22:   a = c;
23:   std::cout << "a.size = " << a.size() << std::endl; // a.size=10 と表
示
24:   for (int i=0; i<a.size(); i++) std::cout << a[i] << " ";
25:   std::cout << std::endl;
26:
27:   a[15] = 13; // 確保していない領域を書き換え
28:   std::cout << a[15] << std::endl; // たまたま正しい値が入っているかのように
見える
29:   std::cout << a.size() << std::endl; // サイズは元のまま
30:
31:   return 0;
32: }
```

- 5: このように<> を用いて型などを渡す構文はテンプレート (template) と呼ばれ, 最近 C++ に導入された.
  - `vector<int>` の int 部分を書き換えれば, 任意の型に対する vector が作れる.
  - 1 章で定義した stack では整数型しか扱えなかったが, テンプレート構文を用いて型汎用のクラスとすれば, 任意の型に対する stack が定義できるようになる.
  - テンプレートは, コンパイル時 (プリプロセッサの処理時) に型を代入したものに展開される.
  - テンプレートの構文の詳細はここでは省略する.
- 17: サイズの小さい vector が代入された場合は, サイズは小さくなる (割り当てられた領域はそのまま).
- 22: サイズの大きい vector が代入された場合, サイズは大きくなる. 割り当てられた領域に収まらない場合は,「再配置」が行われる. 再配置とは, 1) 大きなサイズの配列を新たに割り当て, 2) もとの配列をここにコピーし, 3) もとの配列を解放する, という処理である.

  【注】このため, ポインタを用いた vector の要素の直接のアクセスは行うべきではない.

  【注】再配置が頻繁に行われると非常に効率が悪いので, 要素数の上限があらかじめわかっている場合には, 宣言時にそれだけの領域を確保して再配置を避けるべきである. また, メンバ関数 reserve(int s) を用いると, 再配置を行って要素 s 個分の配列の領域を確保することができる. また, 現在確保されている領域はメンバ関数 capacity() により参照できる.
- 27: a[i] の i がサイズ違反をしていても, チェックは行われず, 思わぬメモリーエラーにつながるのは, 従来の配列と同様である. チェックを行っていないのは, 従来の配列と同レベルのアクセスの高速性を維持するためである. なお, 新しめのコンパイラ (g++ だと 2.90.29 以上) では, resize(int newsize) というメンバ関数で, サイズを変更することができる.

- スタック操作

– vector 型には次のスタック操作が定義されており, vector<T> は, 型 T のデータを保持するスタックとしても使える. いちいちスタッククラスを定義しなくてよいし, 何よりもスタックオーバフローの心配が無い (自動的に再配置が行われる) ので便利である.

1. void push_back(T d) … データ d を末尾に追加
2. T& back() … 末尾のデータにアクセス
3. void pop_back() … 末尾のデータを削除

[List 2.5]

```
 1: #include <iostream>
 2: #include <vector>
 3:
 4: int main() {
 5:   std::vector<int> s; // 初期のサイズ指定を省略すると, サイズ 0 となる
 6:   s.push_back(5);
 7:   s.push_back(7);
 8:   s.push_back(9);
 9:   s.push_back(11);
10:   std::cout << "size = " << s.size() << std::endl;
11:   for (int i=0; i<s.size(); i++) {
12:     std::cout << s[i] << " ";
13:   }
14:   std::cout << std::endl;
15:   std::cout << s.back() << std::endl;
16:   s.pop_back();
17:   std::cout << s.back() << std::endl;
18:   s.pop_back();
19:   std::cout << "size = " << s.size() << std::endl;
20:   for (int i=0; i<s.size(); i++) {
21:     std::cout << s[i] << " ";
22:   }
23:   std::cout << std::endl;
24:   return 0;
25: }
```

– push_back(T) によりサイズは 1 つ増え, 確保した容量を越えると再配置が行われる.

【注】サイズを一つ増やすたびに再配置を行っていたのでは, 明らかに効率が悪い. そこで, g++ の実装では push_back(T) により容量が不足すると, 容量を 2 倍にして再配置を行うようである. (push_back() する毎に capacity() を表示してみると分かる.)

- vector の vector
  - vector をはじめ, 全てコンテナはネスティング可能
  - 整数の vector の vector は, vector<vector<int> > と表せる.

【注】最後の 2 つの“ > ”の間にスペースを入れることが必要. そうでないと, 字句解析系が「シフト演算」(>>) と認識してしまい, 構文エラーとなる.

【EX-2.2】 前の演習【EX-2.1】で作成した [List 2.3] のプログラムを, 通常の配列の代わりに vector<Entry> を用いて書き換えよ. 23〜27 行目のデータの追加には push_back を用いよ.

2.2.2 deque

- deque は「双頭キュー」
  - vector の全機能＋先頭でのデータの出し入れ操作が可能
- 宣言
  - `<deque>` をインクルードする
  - 型 T のデータを保持する deque は `std::deque<T>`. 例えば, 整数を保持する deque は, `std::deque<int> d;` のように宣言する.
- 先頭での操作
  1. push_front(T d) … 先頭に型 T のデータ d を挿入. 他のデータは後ろにシフトされる. すなわち, 新しいデータが a[0] に入り, もとの a[0], a[1], a[2], … はそれぞれ a[1], a[2], a[3], … に移される.
  2. T& front() … 先頭要素にアクセス (これは `vector<T>` でも可能)
  3. pop_front() … 先頭要素を削除. 他のデータは前にシフトされる.

  ☆ push_back() と pop_front() を用いると, 待ち行列 (queue) を実現できる.

【注】push_front(T) および pop_front() を `vector<T>` の持つデータ構造上で実装しようとすると, 要素数 n に対して $O(n)$ 時間がかかってしまう (データのシフトのため) が, `deque<T>` では両方とも $O(1)$ 時間でできる (リングバッファを用いて実装されている). ただし, 要素のアクセス (a[i]) は `vector<T>` よりわずかに遅い (リングバッファ内でのオフセットの計算のため).

2.2.3 list

- list は双方向リスト
  - 先頭と末尾以外の場所でも挿入と削除が可能.
  - [] を用いたランダムアクセスはできない.
- 宣言
  1. `<list>` を include する.
  2. 型 T のデータを保持するリストは `std::list<T>`. 例えば整数データを保持する list は, `std::list<int> a;` のように宣言する.
- iterator (「反復子」と訳される)
  リストの要素を指すための型. ポインタを抽象化したものと言える. ここではひとまず, 「ポインタのようなもの」と考えれば理解できる. `std::list<T>` の iterator は, `std::list<T>::iterator` と書く. 例えば, `std::list<T>` の iterator は,

  ```
  std::list<int>::iterator p;
  ```

  のように宣言する.
- 整数リストにデータを挿入し, 表示する例

[List 2.6]

```
 1: #include <iostream>
 2: #include <list>
 3:
 4: int main()
 5: {
 6:   std::list<int> li;
 7:   li.push_back(3); // 後尾に挿入
 8:   li.push_back(7); // 後尾に挿入
 9:   li.push_front(2); // 先頭に挿入
10:   li.push_front(5); // 先頭に挿入
11:
12:   std::list<int>::iterator p; // iterator p の宣言
13:   for (p=li.begin(); p!=li.end(); p++) { // li の最初から最後まで
14:     std::cout << *p << " "; // p の指すデータを出力
15:   }
16:   std::cout << std::endl;
17:
12:   std::list<int>::reverse_iterator r; // 逆走査用 iterator の宣言
13:   for (r=li.rbegin(); r!=li.rend(); r++) { // li の最後から最初まで
14:     std::cout << *r << " "; //
15:   }
16:   std::cout << std::endl;
17:
18:   return 0;
19: }
```

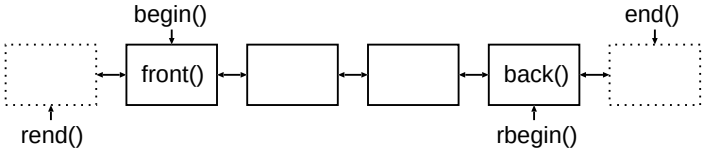
これを実行すると,

```
5 2 3 7
7 3 2 5
```

が出力される. list の使い型について以下を参照.

- データの挿入
  x を型 T のデータ, p, q, r を `std::list<T>` の iterator とする.
  1. push_back(x), push_front(x) … vector や deque と同じ
  2. insert(p,x) … p の指す要素の直前に x を挿入
- データの削除
  1. pop_back(x), pop_front(x) … vector や deque と同じ
  2. erase(p) … p の指す要素を削除
  3. erase(p,q) … p の指す要素から q の指す要素の直前までを削除
  4. clear() … 全要素を削除
- 先頭と末尾のデータの参照

1. front() … 最初の要素
2. back() … 最後の要素
3. begin() … 最初の要素を指す iterator
4. end() … 最後の要素の次を指す iterator ()
5. rbegin() … 逆順でたどるときの最初の要素を指す reverse_iterator (逆順走査用の iterator)
6. rend() … 逆順でたどるときの最後の要素の次を指す reverse_iterator



- iterator
  p が iterator の時

  1. *p … p の指す要素

  【注】データが構造体やクラスのとき, そのメンバー m は (*p).m または `p->m` で表す. ただし, 古いコンパイラでは, 後者はサポートされていない.

  2. p++ … p が次の要素を指すようにする
  3. `p--` … p が前の要素を指すようにする
  4. `std::list<T>` li 中の全データのアクセスは, p を iterator として,

     ```
     for (p=li.begin(); p!=li.end(); p++) { *p = ...}
     ```

     で行える.

- reverse_iterator reverse_iterator は逆順に走査用の iterator. 方向が逆になる以外は通常の iterator と同じ. r を reverse_iterator とすると

  1. *r … r の指す要素
  2. r++ … r が (走査方向に向かって) 次の要素を指すようにする
  3. r-- … r が (走査方向に向かって) 前の要素を指すようにする
  4. `std::list<T>` li 中の全データのアクセスは,

     ```
     for (r=li.rbegin(); r!=li.rend(); r++) { *r = ...}
     ```

     で行える.

  【注】r-- でないことに注意.

【EX-2.3】次のプログラムは, 学生の成績データを管理するものである. クラス Record は, 一人の学生の [番号 (id), 名前 (name), 点数 (score)] を記録するもので, 出力用の関数 (operator<<) が定義される. 成績データは, クラス Seiseki の data に Record のリストとして記録されている. Seiseki は operator<< と 3 つのメンバ関数 insert, lookup, erase_worst を持つ. その働きは,

1. insert(int id, const std::string& nm, int s): [番号 id, 名前 nm, 点数 s] というデータを追加する.
2. lookup(int id): 番号が id のデータを検索し, std::cout に出力する. (なければ not found と出力).
3. erase_worst(): 点数の最も悪かった学生の Record を削除する.

というものである. 空白を埋め, プログラムを完成させよ.

【注】

- リスト中は特にソーティングする必要はないが, 余裕があれば insert の際にリストが番号順にソートされているようにし, lookup の際にリストを最後まで操作しなくても与えられた学生がリスト中に存在しないことを検出できるようにせよ.
- 点数最下位の学生が複数いた場合は, 1 名だけ削除するという仕様でもよいが, 余裕があれば, 同点最下位をすべて削除するようにせよ. この際, リストを走査しながら複数要素を削除して行く操作には注意が必要である. iterator p の指す要素を削除してから p++ を行おうとすると, 既に p の指す要素が消滅しているので, p++ は正常に行われないからである. 簡単に思いつく安全な方法としては, 一旦リストを走査して削除すべき要素の iterator を配列 (vector) に記録してからそれらの要素をまとめて消去する, というのが考えられる.
- リストの 33 行目の p の宣言で, iterator の代わりに const_iterator を用いているが, これは, 「リストの要素を参照するだけで書き換えは行わない」ことを明示的に宣言するものである. iterator のままでも大きな問題は無いが, 関数 operator<<(std::ostream&, const Seiseki&) の第 2 引数が書き換えられる可能性があるとコンパイラが判断するので, 第 2 引数の const 宣言と衝突するというエラーや警告が発せられる. const 宣言をはずしてしまってもよいが, よりクリーンに書くならこのようになる.

[List 2.7]

```
 1: #include <iostream>
 2: #include <string>
 3: #include <list>
 4:
 5: class Record {
 6:   public:
 7:     int id;
 8:     std::string name;
 9:     int score;
10:     Record() {}
11:     Record(int i, const std::string& nm, int s) {
12:       id = i;
13:       name = nm;
14:       score = s;
15:     }
16: };
17:
18: std::ostream& operator<<(std::ostream& os, const Record& r) {
19:   os << "[" << r.id << "] " << r.name << " : " << r.score;
20:   return os;
21: }
22:
23: class Seiseki {
24:   public:
25:     std::list<Record> data;
26:     void insert(int, const std::string&, int);
27:     void lookup(int);
```

```
28:     void erase_worst();
29: };
30:
31: std::ostream& operator<<(std::ostream& os, const Seiseki& s) {
32:   os << "*** Seiseki ***" << std::endl;
33:   for (std::list<Record>::const_iterator p = s.data.begin();
          p != s.data.end(); p++) {
34:     os << *p << std::endl;
35:   }
36:   return os;
37: }
38:
39: void Seiseki::insert(int id, const std::string& nm, int s) {
      この部分を埋める
40: }
41:
42: void Seiseki::lookup(int id) {
      この部分を埋める
43: }
44:
45: void Seiseki::erase_worst() {
      この部分を埋める
46: }
47:
48: int main()
49: {
50:   Seiseki s;
51:   s.insert(7001,"aaaa",89);
52:   s.insert(7123,"bbbb",70);
53:   s.insert(7013,"cccc",55);
54:   s.insert(7200,"dddd",99);
55:   s.insert(7087,"eeee",83);
56:
57:   std::cout << s;
58:
59:   int id;
60:   std::cout << "> ";
61:   std::cin >> id;
62:   while (id!=0) {
63:     s.lookup(id);
64:     std::cout << "> ";
65:     std::cin >> id;
66:   }
67:
68:   s.erase_worst();
```

```
69:   std::cout << s;
70:
71:   return 0;
72: }
```

2.2.4 map

- map は「連想配列」と呼ばれるデータ構造
  - 通常の配列の添字には自然数 (しかもサイズ未満) でなければならないのに対し, 文字列, ポインタ, ユーザ定義のデータなど任意のデータが添字 (キー) として許される. 例えば, a[0], a[1], … だけでなく, a[”Onoye”], a[”Hashimoto”], … などの記法も可能になる.
  - ただし, 当然ながら通常の配列よりはアクセスが遅い. 内部は二色木 (red-black tree) を用いて実装されているので, 格納されているデータ数が n の時のアクセス時間は O(logn) になる. また, キーの比較が定義されていることが必要である.
- 宣言とアクセス
  1. `<map>` を include する.
  2. キー (添字) の型が K で, データ型が T の map は `std::map<K,T>` と書ける. 例えば double をキーとして int 型データを保持する map は, `std::map<double,int>` となる.
  3. map 型の変数を m とすると, キー k によるデータのアクセスは, m[k] で行える.
- double をキーとする連想配列の例

[List 2.8]

```
 1: #include <iostream>
 2: #include <map>
 3:
 4: int main()
 5: {
 6:   std::map<double, int> fmap;
 7:   fmap[80.00] = 21;
 8:   fmap[82.50] = 33;
 9:   fmap[86.66] = 44;
10:
11:   std::cout << fmap[80.00] << std::endl;
12:   std::cout << fmap[82.50] << std::endl;
13:   std::cout << fmap[86.66] << std::endl;
14:   std::cout << fmap[92.13] << std::endl;
   // 未定義の要素に対しては, データのデフォルト値 (int 型の場合は 0) が返される
15:
16:   return 0;
17: }
```

実行すると

```
21
33
44
0
```

が出力される.

- string をキーとする連想配列の例. これが最も有り得るな利用法だが, g++ でも少しバージョンが古いと, string ライブラリのバージョンの問題で, コンパイル出来ないものがある.

```
[List 2.9]
 1: #include <iostream>
 2: #include <map>
 3: #include <string>
 4:
 5: int main()
 6: {
 7:   std::map<std::string,int> semester;
 8:   semester["logic circuit"] = 2;
 9:   semester["compiler"] = 5;
10:   semester["exercise in information systems 2"] = 6;
11:
12:   std::cout << semester["logic circuit"] << "\n";
13:   std::cout << semester["compiler"] << "\n";
14:   std::cout << semester["circuit theory"] << "\n";
15:
16:   return 0;
17: }
```

実行すると

```
2
5
0
```

が出力される.

- データの走査
  map に登録された全データを取り出すことができる.
  - 取り出しは list と同様, map にも iterator, begin(), end() が定義されているので, 同じ構文を用いて行うことができる.
  - ただし, map は, キーとデータのペア (pair) の形でデータを保持しているので, iterator によって取り出したデータのメンバー first がキーを, second がデータを表すところが異なる.
  - 具体的例は下のリスト参照.

```
[List 2.10]
1: std::map<double, int>::iterator p;
2:
```

```
3: for (p=fmap.begin(); p!=fmap.end(); p++) {
4:   std::cout << p->first << ": " << p->second << std::endl;
     // 古いコンパイラでは, (*p).first, (*p).second にする
5: }
```

- データの検索, 消去
  - [List 2.8] で得られた結果に対して次のリストを追加して fmap の中身を表示してみると, 読み出ししか行っていない 92.13 に対してもデータが登録されてしまっていることに気付く. このように, 検索のために [] 演算を使うと, 予期しないデータが書き込まれてしまう.
  - データを追加しないで検索したい場合には, メンバ関数 find(K) (この K はキーの型) を使う. find(K key) はキーが key のデータを map 中に捜し, 見つかればその iterator を, なければ end() を返す.
  - p が iterator の時, メンバ関数 erase(p) は p の指す要素を削除する.
  - fmap において入力したキーのデータを捜し, あればそれを表示して削除するコードは次のようになる.

```
[List 2.11]
1: std::map<double, int>::iterator p;
2: double f;
3: std::cin >> f;
4: if ((p=fmap.find(f))==fmap.end()) std::cout << "not found" << std::endl;
5: else {
6:   std::cout << p->first << ": " << p->second << std::endl;
7:   fmap.erase(p);
8: }
```

【EX-2.4】 [List 2.8], [List 2.10], [List 2.11] の動作を確認せよ. 手持ちのコンパイラが string をキーとする map をコンパイルできるなら, キーを double から string に変えて試してみてもよい.

2.3 共通アルゴリズム

STL は, 前述のようなコンテナのクラスを提供しているだけでなく, そのコンテナに対する, 検索, ソーティングなどの様々なアルゴリズムを提供している.

- 注目すべき点は, これらのアルゴリズムは, 個々のコンテナのクラスのメンバー関数としてクラス毎に一々定義されているのではなく, 全てのコンテナが共通のアルゴリズムで動作するようになっていることである.
- STL コンテナは, 同じインタフェースでコンテナ内データの操作ができるようになっており, アルゴリズムはこれらのインタフェースを利用して作られている.
- さらに, ユーザが STL コンテナと同じインタフェースで新しいコンテナクラスを作れば, これらの共通アルゴリズムを全て使うことができる. C++ のクラスの枠組を用いると, このようなことも可能になるわけである.

2.4 STL コンテナのインタフェース

- iterator によるデータのアクセス
  vector 型の変数 v に対しては, 全データを順にアクセスする方法として, 整数インデックス i を用いる方法

```
for (i=0; i<v.size(); i++){v[i] = ...}
```

を紹介したが, list で紹介した iterator を用いる方法も使える. すなわち, p を v の iterator とすると, 上の文は,

```
for (p=v.begin(); p!=v.end(); p++){*p = ...}
```

と等価である. iterator を用いた操作は全ての STL コンテナに対して共通に用いることができる. (というよりも, 全てのコンテナに対して同じデータ操作のインタフェースを提供するものとして, iterator というもの (実はクラス) が考え出された.)

- その他の演算と計算時間

	vector	deque	list	map
begin, end	$O(1)$	$O(1)$	$O(1)$	$O(1)$
rbegin, rend	$O(1)$	$O(1)$	$O(1)$	$O(1)$
front, back	$O(1)$	$O(1)$	$O(1)$	—
push_back pop_back	$O(1)$	$O(1)$	$O(1)$	—
push_front, pop_back	—	$O(1)$	$O(1)$	—
[]	$O(1)$	$O(1)$	—	$O(\log n)$
insert	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$
erase	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$
size	$O(1)$	$O(1)$	$O(1)$	$O(1)$
==, !=, $<$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

2.4.1 find

- find 関数は検索を行う.
  - 全てのコンテナに対して適用できる. (ただし, map はデータ保持の構造が異なるので, find をメンバ関数として持っている).
  - p1, p2 をコンテナの iterator, d を捜したいデータとすると, find(p1,p2,d) は, p1 から始めて p2 の直前の要素までの中で d を捜す. もし見つかれば, その要素への iterator を返し, 見つからなければ, p2 を返す.

```
[List 2.12]
 1: #include <iostream>
 2: #include <list>
 3: #include <algorithm>
 4:
 5: int main()
 6: {
 7:   typedef std::list<int> int_list;
 8:   typedef int_list::iterator int_list_iter;
 9:   int_list li;
10:   li.push_back(3);
11:   li.push_back(5);
12:   li.push_back(2);
13:   li.push_back(3);
14:   li.push_back(2);
15:   li.push_back(3);
16:
```

```
17:   int_list_iter p=find(li.begin(),li.end(),2);//リスト全体の中で 2 を検索
18:
19:   if (p==li.end()) std::cout << "not found" << std::endl; // なかった場
合の処理
20:   else {
21:     std::cout << *p << " found" << std::endl;
22:     *p = 3; // 2 を 3 に書き換える
23:
24:     p = find(++p, li.end(), 2); // 見つかった点の直後から検索継続
25:
26:     if (p==li.end()) std::cout << "not found" << std::endl;
27:     else {
28:       std::cout << *p << " found" << std::endl;
29:     }
30:   }
31:
32:   return 0;
33: }
```

2.4.2 sort

- sort はソーティングを行う
  - [] によるランダムアクセスが可能なコンテナ (今回紹介したものの中では vector, deque) に適用可能.

    【注】list はランダムアクセスができないため, この標準アルゴリズムは使えないが, 隣接要素の交換に基づくソート sort がメンバ関数として準備されている.
  - p1, p2 をコンテナの iterator, とすると, sort(p1,p2) は, p1 から始めて p2 の直前の要素までの要素をソートする.
  - sort は quick sort で実装されている. このため,

    1. 平均時間は $O(n \log n)$ だが, 理論的には最悪 $O(n^2)$ かかることがあり得る.
    2. 同じ大きさのキーを持つデータの順序は保証されない

    という性質を持つ. 同じキーを持つデータに対してはもとの順序を保存するという性質を持つソートが欲しい場合には stable sort を用いる. stable sort の計算時間は $O(n \log n \log n)$ だが, 十分メモリ領域があれば $O(n \log n)$ に近づくとのこと.

```
[List 2.13]
 1: #include <iostream>
 2: #include <vector>
 3: #include <algorithm>
 4:
 5: int main()
 6: {
 7:   std::vector<int> a;
 8:   a.push_back(12);
```

```
 9:   a.push_back(25);
10:   a.push_back(1);
11:   a.push_back(9);
12:   a.push_back(30);
13:   a.push_back(4);
14:
15:   sort(a.begin(), a.end());
16:
17:   for (int i=0; i<a.size(); i++) {
18:     std::cout << a[i] << " ";
19:   }
20:   std::cout << std::endl;
21:
22:   return 0;
23: }
```

- 降順ソート
  - sort のデフォルトは昇順ソートである.
  - 昇順以外のソートや, 比較が定義されていない場合には, 比較の方法を第 3 引数に与える必要がある. [List 2.13] で降順ソートをする場合は, 15 行目を

    ```
    15': sort(a.begin(), a.end(), std::greater<int>());
    ```

    に書き換えれば良い.

  【注】greater を定義している<functional> を include する必要があるかも知れない.

- ユーザ定義の比較関数
  - 比較関数は「関数オブジェクト」の形で準備して sort の第 3 引数に渡す.

  【注】「関数オブジェクト」の詳細は省略する. ひとまず構文だけ知っていればよい.
  - 【EX-2.2】で作成した Phonebook のデータを, 名前順と電話番号順にソートするコードは次の通り.

```
[List 2.14]
    ...
 1: class by_name { // 名前の比較関数オブジェクト
 2:   public:
 3:     bool operator()(const Entry& e1, const Entry& e2) const {
 4:       return e1.name < e2.name;
 5:     }
 6: };
 7:
 8: class by_phone { // 番号の比較関数オブジェクト
 9:   public:
10:     bool operator()(const Entry& e1, const Entry& e2) const {
11:       return e1.phone < e2.phone;
12:     }
13: };
```

```
14:
    …
15:
16: int main()
17: {
      …
18:   sort(e.begin(),e.end(), by_name()); // 名前でソート
19:   sort(e.begin(),e.end(), by_phone()); // 番号でソート
…
20: }
```

【EX-2.5】 前の演習【EX-2.2】で作成したプログラムにこのコードを追加し, ソートの動作を確認せよ.

おわりに

STL にはここに紹介した以外にも多くのコンテナやアルゴリズムがある. 紙面と演習時間の都合で全て紹介できなかったのが残念だが, 本格的にプログラムを書く時になったら, 本を参照して欲しい.

【謝辞】 本資料は，石浦菜岐佐先生が大阪大学在職中に作成されたものを一部修正したものである.