

## 2. C++ (2) 「標準ライブラリ STL」

♣ 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,などの問題があった.
- C++ では, 文字列のクラスとして様々なものが提案され, 実装されてきた. 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` など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できる.

[List 2.1]

```
1: #include <iostream>
2: #include <string>
3:
4: int main(void)
5: {
6:     std::string message = "Enter strings";
7:     std::cout << message << ":  ";
8:     std::string s;
9:     std::cin >> s;
10:    std::cout << "word 1 = " << s << std::endl;
```

```

11:     std::cin >> s;
12:     std::cout << "word 2 = " << s << std::endl;
13:     getline(std::cin,s);
14:     std::cout << "line = " << "\"" << s << "\"" << std::endl;
15:     getline(std::cin,s);
16:     std::cout << "line = " << "\"" << s << "\"" << std::endl;
17:     return 0;
18: }

```

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.

- 接続, 比較

- 接続は + により行える. += 演算も使える. s+=a で s の末尾に a のコピーが接続される.
- == 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(void)
5: {
6:     std::string a = "C";
7:     std::string b = " ";
8:     std::string c = "programming";
9:
10:    std::string x = a + b + c;           // x = "C programming"
11:    std::string y = a + "++" + b + c;    // y = "C++ programming"
12:    std::string z = a;                   // z = "C"
13:    z += b;                               // z = "C "
14:    z += c;                               // z = "C programming"
15:
16:    std::cout << "x = " << x << std::endl;
17:    std::cout << "y = " << y << std::endl;
18:    std::cout << "z = " << z << std::endl;

```

```

19:
20:     if      (x==y) std::cout << "x==y\n";
21:     else if (x<y)  std::cout << "x<y\n";    // これが成立
22:     else          std::cout << "x>y\n";
23:
24:     if      (x==z) std::cout << "x==z\n";    // これが成立
25:     else if (x<z)  std::cout << "x<z\n";
26:     else          std::cout << "x>z\n";
27:
28:     return 0;
29: }

```

これを実行すると、

```

x = C programming
y = C++ programming
z = C programming
x<y
x==z

```

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* = "fukada kyoko" に対して, *s.substr(4,6)* は "da kyo" 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* = "fukada kyoko" に対して, *s.substr(7)* は "kyoko" 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, 見つければその先頭文字の位置を返す. 見つからなかった場合は "s.npos" (無効な文字位置を表すもの) を返すことになっている. 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.

### 課題 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,

1. 三田の電話番号の市外局番を削除する ("079-xxx-xxxx" を "xxx-xxxx" に書き換える).
2. 入力した文字列を名前に含むものを検索し, その名前と電話番号を出力する.

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] は講義のホームページの「プログラム」のページよりダウンロードできる.

[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(void)
20:     {
21:         Entry e[10];
22:
23:         int n = 0;
24:         e[n++] = Entry("Kwansei Gakuin University (PR)", "0798-54-6017");
25:         e[n++] = Entry("Kwansei Gakuin University (KSC)", "079-565-7600");
26:         e[n++] = Entry("Kobe University", "078-881-1212");
27:         e[n++] = Entry("Sanda Woodytown SATY", "079-564-8800");
28:         e[n++] = Entry("Sanda Hotel", "079-564-1101");
29:
30:         for (int i=0; i<n; i++) {
31:             // -----
32:             // 電話番号 "079-xxx-xxxx" を "xxx-xxxx" にせよ
33:             // -----
34:         }
35:
36:         std::cout << std::endl;
37:         std::cout << "検索用文字列を入力して下さい: ";
38:         std::string s;
39:         std::cin >> s;
40:         // -----
41:         //  name に s を含むデータを全て表示せよ
42:         // -----
43:
44:         return 0;
45:     }

```

☆ いちいち “std::” を書くのが面倒感じる場合は、プログラムの冒頭で

```
using namespace std;
```

と書けば、省略できる。

## 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、ソート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、リストの集合なども扱うことができる。
- STL の利用により、リスト処理などの動的記憶管理や、二分木処理、探索、ソート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(void) {
5:     std::vector<int> a(5);
6:     std::cout << "a.size = " << a.size() << std::endl;
7:     // a.size = 5 と表示
8:     for (int i=0; i<a.size(); i++) a[i] = i;
9:     for (int i=0; i<a.size(); i++) std::cout << a[i] << " ";
10:    std::cout << std::endl;
11:    // 0 1 2 3 4 と表示
12:
```

```

13:     std::vector<int> b(3);
14:     b[0] = 7; b[1] = 5; b[2] = 3;
15:
16:     std::vector<int> c(10);
17:     for (int i=0; i<c.size(); i++) c[i] = i*i;
18:
19:     a = b;
20:     std::cout << "a.size = " << a.size() << std::endl;
21:     // a.size = 3 と表示
22:     for (int i=0; i<a.size(); i++) std::cout << a[i] << " ";
23:     std::cout << std::endl;
24:     // 7 5 3 と表示
25:
26:     a = c;
27:     std::cout << "a.size = " << a.size() << std::endl;
28:     // a.size = 10 と表示
29:     for (int i=0; i<a.size(); i++) std::cout << a[i] << " ";
30:     std::cout << std::endl;
31:     // 0 1 4 9 16 25 36 49 64 81 と表示
32:
33:     a.resize(30);
34:     // a.size = 30 と表示
35:     std::cout << "a.size = " << a.size() << std::endl;
36:     for (int i=0; i<a.size(); i++) std::cout << a[i] << " ";
37:     std::cout << std::endl;
38:     // 0 1 4 9 16 25 36 49 64 81 0 0 0 ... と表示
39:
40:     return 0;
41: }

```

- 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)** と呼ばれる.
    - \* `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.
  - 19: 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).
  - 26: 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` の要素へ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. そこで, 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. 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. 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.
- `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、サイズの変更は `resize(int newsize)` というメンバ関数で行うことができる。

- 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ー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:  main(void) {
5:
6:      std::vector<int> s;
7:      s.push_back(5);
8:      s.push_back(7);
9:      s.push_back(9);
10:     s.push_back(11); // s = (5 7 9 11)
11:
12:     std::cout << "size = " << s.size() << std::endl; // size = 4
13:     for (int i=0; i<s.size(); i++) {
14:         std::cout << s[i] << " ";
15:     }
16:     std::cout << std::endl;
17:
18:     std::cout << s.back() << std::endl; // 11
19:     s.pop_back(); // s = (5 7 9)
20:     std::cout << s.back() << std::endl; // 9
21:     s.pop_back(); // s = (5 7)
22:
23:     std::cout << "size = " << s.size() << std::endl; // size = 2
24:     for (int i=0; i<s.size(); i++) {
25:         std::cout << s[i] << " ";
26:     }
27:     std::cout << std::endl;
28:
29:     return 0;
30: }
```

- `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「右シフト演算子」(>>) と認識するので構文エラーとなる。

**課題 2.2** 前の演習 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。24～28 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、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、`std::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(void)
5: {
6:     std::list<int> li;
7:
8:     li.push_back(3);    // 末尾に挿入 → li = (3)
9:     li.push_back(7);    // 末尾に挿入 → li = (3 7)
10:    li.push_front(2);    // 先頭に挿入 → li = (2 3 7)
11:    li.push_front(5);    // 先頭に挿入 → li = (5 2 3 7)
12:
13:    // 順に要素を表示
14:    std::list<int>::iterator p;
15:    for (p=li.begin(); p!=li.end(); p++) {
16:        std::cout << *p << " ";
17:    }
18:    std::cout << std::endl;        // 5 2 3 7 と表示
19:
20:    // 逆順に要素を表示
21:    std::list<int>::reverse_iterator r;
22:    for (r=li.rbegin(); r!=li.rend(); r++) {
23:        std::cout << *r << " ";
24:    }
25:    std::cout << std::endl;        // 7 3 2 5 と表示
26:
27:    return 0;
28: }
```

- 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

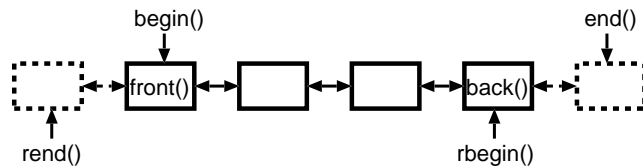
- データの削除

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時、そのメンバー `m` は `(*p).m` または `p->m` で表す。

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。

### 課題 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トする必要はないが、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 37 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. 関数や引数に `const` 宣言をして, このリストの要素を書き換えないことになっている場合には, `iterator` ではなく `const_iterator` を用いる必要がある.

※ [List 2.7] は講義のホームページの「プログラム」のページよりダウンロードできる.

[List 2.7]

```
1: #include <iostream>
2: #include <string>
3: #include <list>
4:
5: // 学生 1 人分の記録
6: class Record {
7:     public:
8:         int id;           // 学籍番号
9:         std::string name; // 名前
10:        int score;        // 点数
11:        Record() {}
12:        Record(int i, const std::string& nm, int s) {
13:            id = i;
14:            name = nm;
15:            score = s;
16:        }
17: };
18:
19: std::ostream& operator<<(std::ostream& os, const Record& r)
20: {
21:     os << "[" << r.id << "]" " << r.name << " : " << r.score;
22:     return os;
23: }
24:
25: // 成績簿 (全学生の記録)
26: class Seiseki {
27:     public:
28:         std::list<Record> data;
29:         void insert(int, const std::string&, int);
30:         void lookup(int) const;
31:         void erase_worst();
32: };
33:
34: std::ostream& operator<<(std::ostream& os, const Seiseki& s)
35: {
36:     os << "*** 成績簿 ***\n";
37:     for (std::list<Record>::const_iterator p = s.data.begin();
38:          p != s.data.end(); p++) {
39:         os << *p << "\n";
40:     }
41:     return os;
42: }
```

```

43:
44: void Seiseki::insert(int id, const std::string& nm, int s)
45: {
46:     // -----
47:     //     この部分を完成させよ
48:     // -----
49: }
50:
51: void Seiseki::lookup(int id) const
52: {
53:     // -----
54:     //     この部分を完成させよ
55:     // -----
56: }
57:
58: void Seiseki::erase_worst()
59: {
60:     // -----
61:     //     この部分を完成させよ
62:     // -----
63: }
64:
65: int main(void)
66: {
67:     Seiseki s;
68:
69:     // 成績の登録
70:     s.insert(7001,"aaaa",89);
71:     s.insert(7123,"bbbb",70);
72:     s.insert(7013,"cccc",55);
73:     s.insert(7200,"dddd",99);
74:     s.insert(7087,"eeee",83);
75:
76:     // 全学生の成績の表示
77:     std::cout << s;
78:
79:     // 入力した id の記録の表示 (0 を入力するまで繰り返し)
80:     int id;
81:     std::cout << "> ";
82:     std::cin >> id;
83:     while (id!=0) {
84:         s.lookup(id);
85:         std::cout << "> ";
86:         std::cin >> id;
87:     }
88:
89:     // 点数が最も悪い学生を消去
90:     s.erase_worst();
91:

```

```

92:    // 全学生の成績の表示
93:    std::cout << s;
94:
95:    return 0;
96: }

```

## 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["Ishiura"]`, `a["Akina"]`, `...` などの記法も可能になる. ただし, 通常の配列よりはアクセスは遅くなる (実装に依存するが).
- 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 `string` をキーとして `int` 型データを保持する map は, `std::map<std::string,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.
- `string` をキーとする連想配列の例.

[List 2.8]

```

1: #include <iostream>
2: #include <map>
3: #include <string>
4:
5: int main(void)
6: {
7:     std::map<std::string,int> semester;
8:     semester["logic circuits"] = 1;
9:     semester["compiler"] = 5;
10:    semester["formal language and automata"] = 3;
11:
12:    std::cout << semester["logic circuits"] << "\n";
13:    std::cout << semester["compiler"] << "\n";
14:    std::cout << semester["network"] << "\n";
15:
16:    return 0;
17: }

```

実行すると

```

1
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 2.9]

```
1:      std::map<std::string,int>::iterator p;  
2:      for (p=semester.begin(); p!=semester.end(); p++) {  
3:          std::cout << p->first << ":  " << p->second << std::endl;  
4:      }
```

#### ● データの検索, 消去

- [List 2.8] に [List 2.9] のコードを追加して stringmap の中身を表示してみると, 読み出ししか行っていない "network" に対してもデータが登録されてしまう。このように, [] 演算は参照に使っただけでデータの書き込みが行われてしまうので, 検索に用いると意図しない結果を引き起こしてしまう。
- 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(key) はキーが key のデータを map 中に探し, 見つければその 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。
- semester において入力したキーのデータを探し, あればそれを削除するコードは次のようになる。

[List 2.10]

```
1:      std::string s;  
2:      std::cout << ">";  
3:      std::cin >> s;  
4:      if ((p=semester.find(s))==semester.end()) {  
5:          std::cout << s << " not found" << std::endl;  
6:      }  
7:      else {  
8:          std::cout << "deleting " << p->first << std::endl;  
9:          semester.erase(p);  
10:     }
```

**課題 2.4** 演習 2.1 で作成した [List 2.3] と同様の動作をするプログラムを map を用いて書け。

map<std::string, std::string> を用い, 名前をキーとして電話番号を求められるようにせよ。

## 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。

### 2.3.1 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_front	—	$O(1)$	$O(1)$	—
[]	$O(1)$	$O(1)$	—	$O(\log n)$
insert	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$
erase	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$
size	$O(1)$	$O(1)$	$O(1)$	$O(1)$
==, !=, <	$O(n)$	$O(n)$	$O(n)$	$O(n)$

### 2.3.2 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,  $d$  を搜したいデータとすると、 $\text{find}(p1, p2, d)$  は、 $p1$  から始めて  $p2$  の直前の要素までの中で  $d$  を搜す。もし見つければ、その要素への iterator を返し、見つからなければ、 $p2$  を返す。

[List 2.11]

```
1:  #include <iostream>
2:  #include <list>
3:  #include <algorithm>
4:
5:  typedef std::list<int> int_list;
6:  typedef int_list::iterator int_list_iter;
7:
8:  int main(void)
9:  {
10:     // 整数リスト li = (3 5 2 3 2 3) を作る
11:     int_list li;
12:     li.push_back(3);
13:     li.push_back(5);
14:     li.push_back(2);
15:     li.push_back(3);
16:     li.push_back(2);
17:     li.push_back(3);
18:
19:     std::cout << ">";
20:     int i;
21:     std::cin >> i;
22:
23:     // リスト中の i を検索
24:     int_list_iter p = find(li.begin(), li.end(), i);
25:     if (p==li.end()) {
26:         // 見つからない場合
27:         std::cout << "not found\n";
28:     }
29:     else {
30:         // 見つかった場合
31:         std::cout << *p << " found\n";
32:
33:         // 見つかったところの直後から更に検索を継続
34:         p = find(++p, li.end(), i);
35:         if (p==li.end()) {
36:             std::cout << "not found\n";
37:         }
38:         else {
39:             std::cout << *p << " found again\n";
40:         }
41:     }
42:
43:     return 0;
44: }
```

**課題 2.5** [List 2.11] のプログラムは、コンテナとして使用している `list` を `vector` に取り換えてもそのまま動く。これを確かめよ。



※ [List 2.11] は講義のホームページの「プログラム」のページよりダウンロードできる。

### 2.3.3 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` では, 同じ大きさのキーを持つデータの順序はどちらになるかわからない。同じ大きさのキーを持つデータに対して元の順序を保存したい場合には `stable_sort` を用いる (計算時間は遅くなる可能性がある)。

[List 2.12]

```
1: #include <iostream>
2: #include <vector>
3: #include <algorithm>
4:
5: int main(void)
6: {
7:     // 整数ベクトル a = (12 25 1 9 30 4)
8:     std::vector<int> a;
9:     a.push_back(12);
10:    a.push_back(25);
11:    a.push_back(1);
12:    a.push_back(9);
13:    a.push_back(30);
14:    a.push_back(4);
15:
16:    // 昇順にソート
17:    sort(a.begin(), a.end());
18:
19:    // 結果の表示
20:    for (int i=0; i<a.size(); i++) {
21:        std::cout << a[i] << " ";
22:    }
23:    std::cout << std::endl;
24:
25:    return 0;
26: }
```

- 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 2.12] 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>());
```

に書き換えればよい.

- 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.

- 課題 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 2.13]

```
...
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(void)
17:  {
18:      ...
19:      sort(e.begin(), e.end(), by_name()); // 名前でソート
20:      sort(e.begin(), e.end(), by_phone()); // 番号でソート
21:      ...
22:  }
```

**課題 2.6** 前の演習 2.2 で作成した `vector` を用いた Phonebook のデータに対し, 名前順のソートと電話番号順のソートを行ってみよ.

## 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. STL は, プログラミングを楽にするものとして画期的であるといえよう.

## 補遺 : auto

- C++11 規格より「型推論」が導入されており, 変数宣言時に型として `auto` を指定すると, コンパイラが適切な型を設定してくれる.

```
std::vector<int> a;  
a.push_back(3);  
a.push_back(5);  
a.push_back(7);
```

のとき

```
for (std::vector<int>::iterator p=a.begin(); p!=a.end(); p++) {  
    std::cout << *p << std::endl;  
}
```

↓

```
for (auto p=a.begin(); p!=a.end(); p++) {  
    std::cout << *p << std::endl;  
}
```

- また, コンテナ `a` に対する繰り返しは下記のように書くこともできる.

```
for (auto&& e : a) {  
    std::cout << e << std::endl;  
}
```

```
for (const auto& e : a) {  
    std::cout << e << std::endl;  
}
```

詳細は省略するが, `auto`, `auto&`, `const auto&`, `auto&&` がある.

- `g++` で C++11 や C++14 規格のコードがコンパイルエラーになる場合, `-std` オプションで規格を指定すれば解決できる (`g++` のバージョンが古すぎなければ).

```
g++ -std=c++11 auto.cpp  
g++ -std=c++14 auto.cpp
```



Nagisa ISHIURA