STL 入門

2012-06-01

はじめに

STL (標準テンプレートライブラリ) は, C++ の標準ライブラリで,

- テンプレートという機能で, 任意の型を扱えるデータ構造 (**コンテナ**) を提供する
- **イテレータ**というポインタの一般化したもので統一的に扱える
- 汎用の**アルゴリズム**が豊富に提供されている

といった特徴を持つ.

STL はオブジェクト指向プログラミングとは全く関係がない. STL はこれまでのプログラミングにおいて行われてきたデータ構造とアルゴリズムを抽象化し, 使いやすくしたものである. また, CGAL など STL の思想に基づいて設計されているライブラリがある.

この資料では, STL のいくつかの例を使って, 基本的な思想と使い方を説明する. なお, 特殊なイテレータや, **関数オブジェクト** などの高度なものについて取り扱わない. これは主に STL のアルゴリズムをカスタマイズする際に使われるものである.

注意　STL はすべて `std` 名前空間に属している. この資料では, すべて `std::` が省略されている.

参考文献

- Standard Template Library プログラミング on the Web
  `http://episteme.wankuma.com/stlprog/`
- Effective STL, スコットメイヤーズ著

1 配列の代わりに vector を使おう

vector

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{

  vector<int> A;


  for(;;)
  {
    int x;
    cin >> x;
    if(cin.fail()) break;

    A.push_back(x);

  }

  const size_t n = A.size();

  cout << n << " values\n";
  for(size_t i = 0; i < n; ++i)
  {
    cout << "[" << i << "] "
                  << A[i] << "\n";
  }
}
```

配列

```
#include <iostream>


using namespace std;

int main()
{
  const size_t MaxSize = 1000;
  int A[MaxSize];

  size_t n = 0; // # of input data
  for(size_t i = 0; i < MaxSize; ++i)
  {
    int x;
    cin >> x;
    if(cin.fail()) break;

    A[i] = x;
    n = i;
  }



  cout << n << " values\n";
  for(size_t i = 0; i < n; ++i)
  {
    cout << "[" << i << "] "
                  << A[i] << "\n";
  }
}
```

要点

- vector 型は, 任意の型の値を格納するデータ構造である. STL に含まれる様々なデータ構造は, **コンテナ** (container) と呼ばれる.
- vector の型は <> で指定する. これは**テンプレート**と呼ばれる C++ の機能を使ったものである.
- vector は, 配列と同様に直列に値を持つデータ構造で, push_back() により値を末尾に追加できる. 例えば, ファイルからデータを読み込む場合を考えると, vector は**自動的に領域が拡大される**ため, 事前にデータの個数を知っておく必要はない. 一方配列を使う場合には, 通常そのデータの個数は不明なので, 適当に大きな領域を確保している.[1]
- vector に含まれる要素数は, size() メソッドで取得できる.
- vector の要素は, 配列と同様に [] 演算子でアクセスできる.

注意 STL を用いたプログラムでは, エラーがあった場合に, **非常に難解なエラーメッセージ**が表示される. これを解読することは困難である. 現在のところ改善策はない.

[1] 個数を数えるために一度データをすべて読み, 配列を確保し, 再度読み込むという方法もあるが, 一般には効率が悪い. また, malloc で動的に確保して, realloc でサイズを変更する方法もある.

2 動的配列より vector を使おう

vector は, メモリ領域の管理が自動で行われるため, new/delete によるメモリの動的確保に比べて安全である.

vector

```
#include <fstream>
#include <vector>

using namespace std;
typedef unsigned char pixel;

int main()
{
  const int W = 640, H = 480;
  vector<pixel> G(W*H);

  {
    ofstream ofs(
         "pattern.pgm", ios::binary);

    // PGM header
    ofs << "P5\n"
      << W << " " << H << "\n255\n";
    // pixel values
    ofs.write(
      (const char*)(&G[0]), W*H);[2]
  }


}
```

動的配列

```
#include <fstream>


using namespace std;
typedef unsigned char pixel;

int main()
{
  const int W = 640, H = 480;
  pixel* G = new pixel[W*H];

  {
    ofstream ofs(
         "pattern.pgm", ios::binary);

    // PGM header
    ofs << "P5\n"
      << W << " " << H << "\n255\n";
    // pixel values
    ofs.write((const char*)G, W*H);

  }

  delete[] G;
}
```

要点

- 用意すべきサイズが既知の場合, vector のコンストラクタによって, あらかじめサイズを指定して確保することができる. 各要素はデフォルトの値 (int のような組み込み型では 0) で初期化される. 初期値を指定する場合は, 第 2 引数を与える.[3]
- 従来のポインタを使った関数に渡すメモリ領域の先頭へのポインタを取得するには &G[0] のように, 0 番目の要素のアドレスを使う.[4] ただし, vector の要素数が 0 の場合には, この式は正しくないので注意.
- vector は自動変数なので, ブロックを抜けると**自動的に削除される**. 動的確保のように自分で解放をする必要はない.

参考 vector は多くの用途で使用可能だが, 配列に比べて若干のオーバーヘッドがある. より高速なアクセスと効率の良いメモリ管理が必要な場合は, <valarray> の valarray が使える. これは STL の一部ではないが, 固定長配列を表現するクラスで, 数学の演算などに特化したメソッドが提供されている. また, C++11 では, STL に固定長配列を表現する array (ヘッダファイルは <array>) が追加された.

[2] ここでは以前の資料と異なり, 簡単のため C スタイルのキャストを使っている.
[3] C++03 までは, 第 2 引数を指定しない場合, デフォルトコンストラクタによって生成されたインスタンスが各要素にコピーされるが, C++11 では, すべての要素をデフォルトコンストラクタで初期化する.
[4] このような, 従来のポインタを用いる関数を**レガシー API** と呼ぶことがある.

3 STL で提供されるコンテナ, アダプタ

順序コンテナ 配列のように順序を持ったデータ構造を扱う.

- <vector>: vector
- <list>: list
- <deque>: deque

連想コンテナ 木構造により, 要素の検索時間が高速に行える.

- <set>: set, multiset
- <map>: map, multimap

コンテナアダプタ 順序コンテナを指定して, スタック及びキューを実現する.

- <stack>: stack
- <queue>: queue, priority_queue

4 順序コンテナの性質比較

各操作による実行時間の比較. ○は定数時間, △は要素数に比例の計算オーダーである. ×は操作が不可能であることを示す.

操作	メソッド	配列	vector	list	deque
インデックスによる要素指定	operator[], at	○	○	×	○
先頭に挿入	push_front	×	△ (insert)	○	○
末尾に挿入	push_back	×	○	○	○
任意の場所に挿入	insert	×	△	○	△
先頭要素を削除	pop_front	×	△ (erase)	○	○
末尾要素を削除	pop_back	×	○	○	○
任意の要素を削除	erase	×	△	○	△

5 コンテナ共通の操作

- コピーコンストラクタ, コピー代入演算子 operator=()
- 先頭および末尾の直後の要素を指すイテレータの取得 begin(), end()
- 比較演算子 operator==(), operator!=(), operator<(), operator<=(), operator>(), operator>=()
- 入れ替え swap()
- サイズの取得 size(), 空であることの判定 empty()

6 イテレータ

イテレータ (反復子) は, ポインタを抽象化したもので, 単なるメモリ領域のアドレス値ではなく, データ構造に応じた移動の振る舞いを持ったオブジェクトである. これにより, ループなどの反復計算が一般化されている.

- ポインタとイテレータの比較
  - デリファレンス
  - 移動
  - 比較
- `begin()` と `end()`
- 無効になる場合
- 入出力ストリームイテレータ
- 挿入イテレータ (インサータ)

7　アルゴリズムの例

アルゴリズムは <algorithm> で宣言されている. 以下にいくつかの例をあげる.

アルゴリズムによる操作	インデックスによる操作

`// 値 x で埋める`
`fill(v.begin(), v.end(), x);`
	`for(int i = 0; i < v.size(); ++i)`
	`{`
	`  v[i] = x;`
	`}`

`// 指定した関数の結果で埋める`
`// need <cstdlib> for std::rand()`
`generate(v.begin(), v.end(), rand);`
	`// need <cstdlib> for std::rand()`
	`for(int i = 0; i < v.size(); ++i)`
	`{`
	`  v[i] = rand();`
	`}`

`// 値が x である要素の個数を数える`
`int n = count(v.begin(), v.end(), x);`
	`int n = 0;`
	`for(int i = 0; i < v.size(); ++i)`
	`{`
	`  if(v[i] == x) ++n;`
	`}`

`// 最大値, 最小値を見つける`[5]
`int M = *max_element(`
`  v.begin(), v.end());`
`int m = *min_element(`
`  v.begin(), v.end());`
	`int M = v[0]; int m = v[0];`
	`for(int i = 1; i < v.size(); ++i)`
	`{`
	`  if(M < v[i]) M = v[i];`
	`  if(m > v[i]) m = v[i];`
	`}`

通常の配列に適用する場合　通常の配列はポインタで操作できるが, ポインタはイテレータとして働くため, 配列にもアルゴリズムが適用できる. 配列名が先頭のイテレータ[6], 配列名に要素数を足したものが末尾のイテレータに相当する. 例えば, `fill` は以下の様になる.

```
const int n = 100;
int A[n];
fill(A, A + n, 255);
```

注意　アルゴリズムには, 関数を指定してそれぞれの要素に適用するものがあるが, そのコードを書くのは多少難しい. C++11 では, ラムダ関数という新しい文法が導入されたため, 比較的簡単にかけるようになった. この資料では説明しない.

[5] `std::max_element()`, `std::min_element()` は, 最大・最小値ではなく, その位置のイテレータを返す. 従って, その値を取るには, `*` によるデリファレンスが必要. `std::max()`, `std::min()` という似た名前の関数があるが (次節参照), これは与えた二つのうちの最大, 最小を返すものであることに注意. なお, C++11 では最大値と最小値を同時に取得する `minmax_element` が追加されている.

[6] 配列名は配列の先頭要素へのポインタ, つまり `&A[0]` に自動的に変換されるため.

8 アルゴリズムの一覧

以下の関数はすべて<algorithm> で宣言されている.[7] 詳細はリファレンスなどを参照すること. なお, 青で強調したものは, イテレータを使わないので, STL の一部として考えなくても有用である.

要素を書き換えない操作

- for_each
- find, find_if, find_end, find_first_of, adjacent_find
- count, count_if
- mismatch
- equal
- search, search_n

要素を書き換える操作

- copy, copy_backward
- swap[8], swap_ranges, iter_swap
- transform
- replace, replace_if, replace_copy, replace_copy_if
- fill, fill_n
- generate, generate_n
- remove, remove_if, remove_copy, remove_copy_if
- unique, unique_copy
- reverse, reverse_copy
- rotate, rotate_copy
- random_shuffle
- partition, stable_partition

並べ替えなどの操作

- sort, stable_sort
- partial_sort, partial_sort_copy
- merge, inplace_merge
- includes, set_union, set_intersection, set_difference, set_symmetric_difference
- push_heap, pop_heap, make_heap, sort_heap
- min, max, min_element, max_element
- lexicographical_compare
- next_permutation, prev_permutation

9 数値配列用に設計されたアルゴリズム

以下の関数は <numeric> で宣言されている.

- accumulate
- inner_product
- adjacent_difference
- partial_sum

[7] このリストは, C++03 のものである. C++11 では大幅に追加されている.
[8] C++11 では, <utility> に移動した.

10 課題

1. §1 を実行しなさい. 以下の 2 通りで試してみること.

   キーボードから入力する. 単純に実行すると, キーボードの入力待ちになる. 空白 (スペース, タブ, または改行) で区切った値をいくつか入力する. `Ctrl-D` で標準入力を終了できる.

   テキストファイルをリダイレクトで読み込ませる. 実行ファイル名を `prog`, 空白区切りのデータを書き込んだテキストファイルを `data.txt` とすると, 以下のように実行すれば良い.

   ```
   $ ./prog < data.txt
   ```

   例では, 値を格納する変数が `int` 型なので, 数値でない文字列を入力するとループが終了する.

2. 以前作成した PGM 画像を書き出すプログラムを以下の様に変更しなさい.

   (a) グレイスケール画像を動的配列ではなく `vector<pixel>` で保持する.

   (b) すべての関数を参照 `vector<pixel>&` で受け取る (ポインタでもよい).

   なお, 変更の必要のない引数 (`const pixel*` で渡していたもの) に対しては, `vector<pixel>` を直接値渡しにすることもできる. しかし, この場合には大きな領域のコピーが発生し, 計算時間が増加するため, 一般的には参照渡しをする方が良い. 非常に小さなオブジェクト (座標など) では値渡しをすることもある.

3. 前回作成した画像処理関数を STL のアルゴリズムを用いて実装しなさい.

   - `fill`: `std::fill()`
   - `flipV`: `std::swap_ranges()`
   - `flipH`: `std::reverse()`
   - 【難問】 `invert`: `std::for_each()`, 反転用の関数オブジェクト[9]

[9] このアルゴリズムには 1 引数を取る**関数オブジェクト**を与えるが, そのようなものの作り方はこの資料では説明していない. また, C++03 までと C++11 以降では作り方に差異がある.