STL の説明

東京大学理学部物理学科 3 年 81540
角田 佑介

平成 21 年 3 月 14 日

1　このテキストの読み方

このテキストの前半は、独学用にきちんと説明を書いたものです。また、後半は要点をまとめたもので、初めて読むには説明不足ですが、軽く復習したいときに向いていると思います。終わりには、このテキストを書くにあたって参考にした URL を載せておきます。このテキストには高度なことや細かいことは書いていないので、そのようなことを知りたいときは参考にして下さい。[1]

2　独学用の説明

2.1　STL とは？

STL はどんなときに役立つかというと、たくさんのデータを取り扱う必要があるときに役立ちます。たくさんのデータを取り扱うためのものといえば、配列が既にあります。しかし、配列をうまく使うのは大変です。例えば、一度決めた要素数を変更することができません。また、配列の要素数を変数で指定したいと思ったら、`p=new int[n];` のようにする必要がありますが、使い終わる度に `delete[] p;` しなければなりません。STL を使えば、配列よりもっと便利にデータを取り扱うことができます。

次に、STL で重要な用語を簡単に説明します。[2]そもそも、STL が何の略語かというと、Standard Template Library の略です。Standard は標準、Library はライブラリなので意味は分かると思いますが、Template とは何でしょう。C++のゼミでは扱いませんでしたが、C++にはテンプレートという機能があります。テンプレートは、簡単に言えば、任意の型を同じように取り扱うための機能です。任意の型に対して配列が存在するように、任意の型に対してデータを扱うための関数やクラスを用意しなければなりません。それが、テンプレートを使うことで可能になります。[3]

データを格納するためのものが、コンテナです。配列のようなものですが、配列より高機能で、用途に応じて様々なものがあります。

配列で言えばポインタに相当するものが、イテレータ (反復子) です。コンテナの要素を指定するのに用います。

---

[1] このテキストに関数のリストとかプログラム例とかを細かく付けてはいないので、そういうものが知りたいときにネットを活用してください。

[2] 詳細な説明はこの後で。

[3] 任意の型が扱える利点は、自分で定義したクラスも同じように取り扱えることです。また、大小関係を定義しておけばソートができますし、同値関係を定義すれば検索ができます。

コンテナのデータを操作したいとき、基本的な機能はコンテナごとに定義されています。しかし、より高度な機能はコンテナごとには定義されていません。その代わりに、イテレータを用いて高度な機能を実現します。このような、イテレータを通じて操作を行う関数をアルゴリズムといいます。なぜイテレータを用いるかというと、一つのアルゴリズムで様々なコンテナのデータを操作することができるからです。[4]

アルゴリズムを用いるときに、何か関数を指定したいことがあります。このようなときは関数ポインタを用いるのもいいですが、STL では関数オブジェクトというものを用いることもあります。[5]ただし、このテキストではあまり詳しくは説明しません。[6]

それでは、具体的な使い方などを見ていきましょう。

2.2 テンプレート

STL に入る前に、テンプレートについて簡単に説明しておきます。時間がないなら、飛ばしても構いません。飛ばしても STL を使うことはできます。

例えば、二つの同じ型の引数の和を計算する関数を書きたいとしましょう。int 型については、次のように書けます。

```
int f(int a, int b){
  return a+b;
}
```

double 型についても作りたいと思ったら、int を double に書き換えれば作れます。しかし、この関数を任意の型について作りたいと思ったら、一つ一つ書くわけにも行きません。そんなときには、テンプレートを使います。書き方は次のようになります。

```
#include <iostream>
using namespace std;

template<class T>          // T が任意の型となります。
T f(T a, T b){
  return a+b;
}

int main(){
  double a=1,b=2.0;
  cout << f(a,b) << endl; // a,b が double 型なので、T は double と判断されます。
  cout << f<double>(1,2.0) << endl; // T が分からないので、T を指定します。
  return 0;
}
```

[4]一つのものが様々なものに対応しているということは、それだけ拡張性が高いということです。新たなコンテナを作れば既存のアルゴリズムが使えますし、新たなアルゴリズムを一つ作れば様々なコンテナに対して用いることができます。

[5]関数オブジェクトを用いた方が速くなることが多いらしいです。

[6]そういえば、2 年生は関数ポインタを教わってません。詳しく知りたい人は、自力で調べてください。3 年生の宿題のページ (http://myoga.web.fc2.com/mayfes/ode/ode_d1.htm) にも書いてあります。

T の部分が、任意の型になります。関数を呼び出すときは、引数の型で T が何かを判断してくれます。ただし、T が判断できないときは、関数名の後に<型名>をつけて T を指定します。任意の型を複数用いるときは、template<class T1, class T2>のように書きます。T1 や T2 を指定するには、<int,double>のように書きます。クラスを定義するときも、同様にしてテンプレートを用いることができます。型を指定するときには Class1<int>のように、クラス名 (Class1) の後に<型名>をつけてください。

2.3 コンテナ

コンテナにはいくつかの種類があるので、一つ一つ紹介していきます。すべての機能をこのテキストで紹介することはできないので、使い方の基本が分かるように機能をほんの少しだけ紹介します。

2.3.1 vector

vector は通常の配列に似たコンテナですが、要素数を自由に変えることができます。[7]使い方の例を少し見てみましょう。

```
#include <vector>           // vector を使うときは vector をインクルードします。
#include <iostream>
using namespace std;

int main(){
  vector<int> vec1;         // int 型 vector の変数を宣言します。

  vec1.push_back(0);        // push_back で末尾に要素を追加します。
  vec1.push_back(-1);
  vec1.push_back(6);

  cout<<vec1.size()<<endl; // size で要素数を求めます。

  for (int i=0; i<vec1.size(); i++) cout<<vec1[i]<<" ";
  cout<<endl;               // 要素は配列と同じように使えます。

  vector<int> vec2(vec1);  // vec2 を宣言し、vec1 の中身をコピーします。

  vec1.pop_back();          // pop_back で末尾の要素を取り除きます。
  vec1[0]=4;                // 要素に代入することもできます。
  for (int i=0; i<vec1.size(); i++) cout<<vec1.at(i)<<" ";
  cout<<endl;               // [] の代わりに at を使うこともできます。
```

[7]要素数が自由に変えられる配列を動的配列と言います。内部では、要素数が変わったときにメモリを新たに確保するのを自動的にやってくれます。

```
  vec1.clear();             // vec1 の要素をすべて消します。
  cout<<vec1.size()<<endl;
  for (int i=0; i<vec2.size(); i++) cout<<vec2[i]<<" ";
  cout<<endl;

  return 0;
}
```

出力は以下のようになります。

```
3
0 -1 6
4 -1
0
0 -1 6
```

vector の宣言は、vector<型名> 変数名のように行います。[8]vector の要素は、配列のように [] で扱えます。また、vector のメンバ関数を用いて色々な操作が行えます。イテレータを用いた機能については、イテレータの節で説明します。

2.3.2 deque

deque は、両端キューと呼ばれるものです。vector では末尾に要素を足したり消したりできるのに対し、deque では先頭と末尾に要素を足したり消したりできます。[9]deque を使うときは、deque をインクルードして下さい。

2.3.3 list

list は、双方向リストと呼ばれるものです。vector や deque では要素が連続して記憶されているのに対し、list では各要素が別々に記憶されています。そして、list の各要素は自分の前後の要素がどこにあるかを記憶しています。[10]vector や deque で途中に要素を足したり消したりすると、他の要素をずらす必要があるので手間がかかります。一方、list で途中に要素を足したり消したりするには、前後の要素の位置の情報を書き換えるだけでいいので簡単です。[11]しかし、list で途中の要素の場所を調べるには、端から一つ一つたどっていく必要があります。一方、vector や deque では途中の要素の場所が簡単に分かります。

list を使うときは、list をインクルードして下さい。

[8]テンプレートクラスの変数を宣言するのと同じ方法です。
[9]途中に要素を足したり消したりすることもできますが、少し時間がかかります。
[10]例えるなら、list は各要素がばらばらに置かれていて、それぞれが順番に糸でつながっているようなものでしょうか。
[11]同じ例えで言うのなら、要素を動かさなくとも前後の糸をつなぎかえるだけでいいということです。

2.3.4 set、multiset

set は集合という名が表しているように、要素の順番を指定することができません。要素が自動的にソートされて並ぶので、要素の検索に適しています。[12]set では要素の重複が許されず、multiset では要素の重複が許されます。set や multiset を使うときは、どちらも set をインクルードして下さい。

2.3.5 map、multimap

map は写像という名が表しているように、キーと値の組を記憶しています。要素はキーの順でソートされているので、キーを検索して対応する値を探すのに適しています。[13]宣言は、`map<`キーの型 , 値の型`>`のようにして行います。要素はキーと値の組[14]となっているので、要素の扱いが少し特殊になります。map ではキーの重複が許されず、multimap ではキーの重複が許されます。[15]map や multimap を使うときは、どちらも map をインクルードして下さい。

2.3.6 コンテナに類するもの

stack(スタック) は、末尾から要素を入れ、末尾から要素を取り出すデータ構造です。queue(キュー) は、末尾から要素を入れ、先頭から要素を取り出すデータ構造です。priority_queue(優先順位付きキュー) は、末尾から要素を入れると自動的にソートされ、先頭から要素を取り出すデータ構造です。これらは vector や deque の機能を制限して作られていて、イテレータを用いることはできません。

bitset はビットの配列で、1 ビットの情報を複数扱うことができます。[16]イテレータは使えません。

string は文字列を表わすデータ構造で、イテレータを使うことができます。[17]通常の文字列は char 型の配列なので取扱いが厄介ですが、string はより使いやすくなっています。

2.4 イテレータ

イテレータには種類があり、種類によって機能が異なります。入力イテレータは、一つ先に進むことと要素を読むことしかできません。出力イテレータは、一つ先に進むことと要素を書くことしかできません。前方イテレータは、一つ先に進むことと要素の読み書きができます。双方向イテレータは、一つ前後に進むことと要素の読み書きができます。ランダムアクセスイテレータは、ポインタのように足し引きができ、要素の読み書きができます。list、set、map のイテレータは双方向イテレータで、vector、deque のイテレータはランダムアクセスイテレータです。

イテレータを用いた例を挙げます。

---

[12]自動ソートのために、内部では木構造を用いているそうです。

[13]map の場合はコンテナ名 [キー] とすることで、キーに対応する値を得ることができます。例えば、map1 という名前の map にキーが”Cirno”、値が”9”という組を登録しておくと、`cout << map1["Cirno"] << endl;` で 9 が出力されます。代入も可能で、`map1["Rumia"] = 10;` とやると、キーが”Rumia”、値が”10”という組が登録されます。

[14]組を表わすために、pair というクラスが STL で定義されています。

[15]multimap では、`[]` を使うことはできません。

[16]bool 型のコンテナよりメモリが節約できます。

[17]wchar_t を用いた wstring というものもあります。

```
#include <vector>
#include <iostream>
using namespace std;

int main(){
  vector<int> vec;
  vector<int>::iterator it; // イテレータを宣言します。

  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);

  it = vec.begin();   // イテレータの指す先を vec の先頭にします。
  cout<<*it<<endl;    // イテレータはポインタと同じように*が使えます。

  it+=2;              // イテレータを 2 つ進めます。
  cout<<*it<<endl;

  it--;               // イテレータを 1 つ戻します。
  it = vec.erase(it); // イテレータの指す要素を取り除きます。
                      // erase のあと it はそのままでは使えないので、
                      // erase の戻り値を代入します。
  it.insert(it,4);    // イテレータの指す先に 4 を挿入します。
                      // insert のあとも it は使えなくなりますが、
                      // あとで it に値を代入するのでそのままにしてあります。

  for(it=vec.begin(); it!=vec.end(); it++) cout<<*it<<" ";
  cout << endl;       // end でイテレータの指す先は、末尾の 1 つ先です。

  return 0;
}
```

出力は以下のようになります。

```
1
3
1 4 3
```

イテレータの型名は、コンテナ名<型名>::iterator となります。[18]コンテナに要素を足したり消したりすると、それまでのイテレータが使えなくなることがあるので注意して下さい。[19]erase は取り除いた要素の位置のイテレータを返すので、これを用いています。始まりを示すイテレータは先頭の要素を指しているのに対し、終わりを示すイテレータは末尾の一つ先の要素を指していることに注意して下さい。

[18]クラスの中で宣言された関数を外で定義するときは、クラス名::関数名とします。これと同じように、クラスの中で宣言された型を外から用いるときは、クラス名::型名となります。

[19]これは、要素数が変わると値を記録しているアドレスが変わることがあるからです。

2.5 アルゴリズム

アルゴリズムを用いた例を挙げます。

```
#include <vector>
#include <algorithm> // アルゴリズムを用いるときは algorithm をインクルードします。
#include <iostream>
using namespace std;

int main(){
  int n;
  vector<int> vec(10, 0);                 // 要素数 10 とし、値 0 で初期化します。
  vector<int>::iterator it, it1, it2;
  for(it=vec.begin(); it!=vec.end(); it++) cout<<*it<<" ";
  cout << endl;                           // 値の表示。

  vec[4]=1;
  for(it=vec.begin(); it!=vec.end(); it++) cout<<*it<<" ";
  cout << endl;                           // 値の表示。

  it1 = find(vec.begin(), vec.end(), 1); // vec から 1 を探します。
  it2 = find(vec.begin(), vec.end(), 2); // vec から 2 を探すが見つかりません。
                                          // このとき it2 は vec.end() になります。
  fill(it1, it2, 3);                      // it1 から it2 の一つ前までを 3 にします。
  for(it=vec.begin(); it!=vec.end(); it++) cout<<*it<<" ";
  cout << endl;                           // 値の表示。

  n = count(vec.begin(), vec.end(), 3);  // vec から 3 の要素数を数えます。
  cout << n << end;

  return 0;
}
```

出力は以下のようになります。

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 3 3 3 3 3 3
6
```

アルゴリズムを使うときは、algorithm をインクルードします。イテレータの引数があることを除けば、アルゴリズムは普通の関数と同じです。[20]

[20]要素の範囲を指定するのにイテレータを 2 つ指定することがありますが、範囲は始まりを表わすイテレータの指す要素から、終わりを表わすイテレータの指す一つ前の要素までです。

2.6 関数オブジェクト

関数を呼び出すときの () は、関数呼び出し演算子という演算子です。なので、クラスの定義で関数呼び出し演算子をオーバーロードすると、そのクラスのオブジェクトを関数のように扱うことができます。そのため、このようなオブジェクトのことを関数オブジェクトといいます。

アルゴリズムの中には、関数オブジェクトを引数とするようなものが存在します。しかし、関数ポインタでも代用可能なので、ここでは関数ポインタを用います。使い方の例は、次のようになります。

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

bool even(int n){
  return n%2 == 0; // n が偶数なら true、奇数なら false となります。
}

int main(){
  vector<int> vec;

  vec.push_back(1);
  vec.push_back(3);
  vec.push_back(4);
  vec.push_back(1);
  vec.push_back(3);
  vec.push_back(9);
  vec.push_back(8);

  cout<<count_if(vec.begin(),vec.end(),even)<<endl;
                    // even の返り値が true となる要素の数を数えます。

  return 0;
}
```

出力は以下のようになります。

```
2
```

関数ポインタを知らない人は、とりあえず関数名を引数にする関数が作れると思っておいて下さい。

3　復習用の簡単な説明

コンテナのメンバ関数やアルゴリズムの一覧とかはありません。ネットを参考にして下さい。

3.1　STL とは？

- STL で、たくさんのデータを楽に取り扱うことができる。
- テンプレートは、任意の型に対する関数やクラスを作るための方法。
- コンテナは、データを格納するためのもの。
- イテレータ (反復子) は、ポインタのようなもの。
- アルゴリズムは、イテレータを引数とする関数。
- 関数オブジェクトは、関数呼び出し演算子をオーバーロードしたオブジェクト。

3.2　テンプレート

- 関数やクラスの定義の前に `template<class T>`を付けると、
- 任意の型 `T` に対する関数やクラスが定義できる。
- T を指定するには、関数名やクラス名の後に<型名>を付ける。

3.3　コンテナ

- コンテナの宣言は、コンテナ名<型名> 変数名のように行う。
- コンテナを用いるときは、コンテナ名のヘッダファイルをインクルード。
- コンテナのメンバ関数を用いて、基本的な操作が行える。
- vector(動的配列) は、要素数が自由に変えられる配列。
- deque(両端キュー) は、両端に要素を足したり消したりするのに向いている。
- list(双方向リスト) は、途中に要素を足したり消したりするのに向いているが、いきなり途中にアクセスするのに向かない。
- set(集合) は、要素の検索に適している。
- map(写像) は、キーを検索して対応する値を探すのに適している。
- multiset や multimap では、要素やキーの重複が許される。

3.3.1 コンテナに類するもの

- stack(スタック) は、末尾から要素を入れ、末尾から要素を取り出す。
- queue(キュー) は、末尾から要素を入れ、先頭から要素を取り出す。
- priority_queue(優先順位付きキュー) は、末尾から要素を入れ、優先順位が最高の要素を取り出す。
- bitset は、1 ビットの情報を複数扱える。
- 上の 4 つでは、イテレータは使えない。
- string(文字列) では、イテレータが使える。通常の文字列より使いやすい。

3.4 イテレータ

- 入力イテレータは、一つ先に進む+要素を読む。
- 出力イテレータは、一つ先に進む+要素を書く。
- 前方イテレータは、入力イテレータ+出力イテレータ。
- 双方向イテレータは、前方イテレータ+一つ後に戻る。
- ランダムアクセスイテレータは、双方向イテレータ+任意の数進む。
- list、set、map は双方向イテレータ、vector、deque はランダムアクセスイテレータ。
- イテレータの型名は、コンテナ名<型名>::iterator。
- コンテナに要素を足したり消したりすると、それまでのイテレータが使えなくなることがある。
- 始まりを示すイテレータは先頭の要素を指し、終わりを示すイテレータは末尾の一つ先の要素を指す。

3.5 アルゴリズム

- アルゴリズムを用いるときは algorithm をインクルード。
- アルゴリズムを用いて、より高度な操作が行える。

4 参考 URL

計算機班の C++のページ (http://myoga.web.fc2.com/mayfes/cpp/) の下にもリンクの貼ってある、以下のページを主に参考にしました。URL は STL のページのものです。

- Programing Place(http://www.geocities.jp/ky_webid/cpp/library/index.html)
- 目指せプログラマー！(http://www5c.biglobe.ne.jp/~ecb/cpp/cpp00.html)

5 補足

VisualC++ 6.0 を使っている人は、そのままコンパイルするとたくさん警告が出ることがあります。警告が出ても害はありませんが、あまり出てほしくはありません。そこで、ソースコードの冒頭に次の行を追加しましょう。

```
#pragma warning(disable:4786) // 警告を出さないための「おまじない」です。
#include <vector>             // include より上に書いて下さい。
```

一行目を書くことで、警告が出なくなります。[21]それでも警告が出るようなら、自分のプログラムが間違っていないか確認しましょう。

6 更新履歴

- 2009/3/14 誤解を招きやすいところを修正。脚注を追加。
- 2009/3/13 VisualC++ 6.0 を使っている人への補足を追加。
- 2009/3/13 初版公開。

[21]意味は分からなくていいけど書かなくてはいけない命令のことを、情報科学の業界では「おまじない」と言ったりします。