データ構造と標準テンプレートライブラリ (STL)
～STL を使えばプログラムはこんなに楽になる？～

筑波大学情報学群情報メディア創成学類 藤澤誠

1　データ構造 (deta structure) とは

プログラムを構成する要素として，古くから言われてきたのが「アルゴリズム (algorithm)」と「データ構造 (deta structure)」である．

2　STL とは

STL とは，**標準テンプレートライブラリ** (Standard Template Library) です．といっても，何のことだと思われるでしょうが，これは，**テンプレートクラス**というものを使った，クラスライブラリのことです．テンプレートクラスについては，2.1 節で説明しますので，ここでは，STL とはなんなんだということを解説します．

STL とは，簡単に言えば，よく使用されるアルゴリズムとデータ構造を実装したライブラリのことです．例えば，動的な配列を実現する**ベクタ**，**リスト**，**キュー (待ち行列)**，**スタック**，そしてそれらをソート，検索，変換する各種のルーチンなどが定義されています．そして，これらの関数，クラスはすべてテンプレートという仕組みで構成されているため，ほとんど任意のデータ型に適用することができます．

実際に例をみてみましょう．

```
vector<int> x;
for(int i = 0; i < 10; ++i){
  x.push_back(i);
}
```

ここで，`vector` というのが動的配列を扱う STL のベクタであり，`<>`で囲んだ型のベクタを定義しています．宣言時に，必要な要素数を定義していないことに注目してください．`x.push_back(i)` としていくことで，勝手に必要な分のメモリを確保していってくれます．

もちろん，これらの機能は，これまでやってきたクラスを使えば，自分で実装することはできます．しかし，STL は様々な人たちによって改良されてきており，ほとんどの場合，自分で作るより高速で効率的です．

2.1　テンプレートクラス

C++(前半) で作成した `Vec3` クラスを思い出してください．あのときは，各次元の要素を `float` 型で定義しました．

```
class Vec3
{
private:
  float x, y, z;
....
};
```

しかし，もし，もっと精度が必要だった場合，double 型や long double にしたいかもしれません．また，自分で定義したクラス型にしたい場合もあります．これらをわざわざ作り直すのは面倒です．このようなときにテンプレートクラスが使えます．

```
template<class Type>
class Vec3
{
private:
  Type x, y, z;
....
};
```

これを使うときには，

```
Vec3<double> v;
```

とすると，メンバ変数 x,y,z は double 型になります．
同様のことが関数についてもでき，それをテンプレート関数と呼びます．

3 vector

さて，STL の中でもみなさんがもっともよく使うであろうものが vector でしょう．vector は，前節のテンプレートクラスで定義された動的配列用の型です．よって，どんな型のベクタでも作れます．

```
vector<int> a;
vector<double> b;
vector<Triangle> c;
```

3 行目のように，自分で作ったクラスも型として使うことができます．また，vector は複数のコンストラクタを持ちます．よって，List.1 のように，初期化の仕方にいくつかの方法があります．

List.1

```
#include <iostream>
#include <vector>
using namespace std;
int main(void)
{
  vector<int> a;        //空のベクタ
  vector<int> b(5, 1); //（サイズ，初期値）
  vector<int> c(b);     //他のベクタによる初期化

  int i, n;
  cout << "n : " << endl;
  cin >> n;
  for(i = 0; i < n; ++i){
    a.push_back(i);
  }
  for(i = 0; i < n; ++i){
    cout << "a[" << i << "] = " << a[i] << endl;
  }
}
```

vector は，先ほども示したように動的な配列です．しかも，配列のメモリ領域は自動的に確保してくれます．List.1 では，push_back メンバ関数を用いて，要素を末尾に追加しています．ここで，vector<int> a は，配列の要素数を指定していないことに注意してください．push_back を使ってデータを配列に入れていくと，vector クラス内で自動的に必要なメモリを確保していってくれます．要素の追加には，push_back の他に insert というのもあります．これは，配列の任意の場所に要素を追加するものです．

また，new の時と違って，delete も必要ありません．なぜなら，vector は必要が無くなったら勝手に破棄してくれるからです．

List.1 の実行結果を以下に示します．n の値はキーボードから入力したものです．

```
n : 5
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
```

3.1 メモリの手動確保

vector は，自動的にメモリを確保してくれる動的配列であると説明しました．しかし，これだけではプログラムする上でまずいことが起こる場合があります．例えば，以下のリストを見てください．

```
vector<int> a;
int* pa;
```

```
int i;
for(i = 0; i < 50000; ++i) a.push_back(i);
pa = &a[3];
for(i = 50000; i < 100000; ++i) a.push_back(i);
cout << *pa << endl;
```

このコードを実行した場合，環境にもよるでしょうが，ほとんどの場合，メモリエラーとなるでしょう．これは，メモリ確保時の再配置によって，a[3] の要素を格納したメモリ領域が移動したため，ポインタ pa の指すアドレス領域に何もなくなってしまったのが原因です．

このように，特に vector 要素に対するアドレスを用いる場合に不都合が起こることがあります．これを防ぐために，あらかじめ要素を確保しておく方法があります．vector では，メモリの手動確保の方法は二つあります．reserve と resize です．

List.2 にその使い方を示します．List.2 では，capacity 関数と size 関数も用いています．両方ともメモリ領域のサイズを返す関数なのですが，意味が違います．まずは，この違いから説明します．size 関数は，実際に値が代入されている，もしくは値を代入することができる要素数を返します．これは，例えば，配列 int t[10]; の 10 にあたる数です．そして，capacity 関数は，実際にはまだベクタ要素として用いることはできないけれど，後で要素数が増えたときのための予約領域として確保されているメモリ領域を返します．

これらをふまえて，List.2 の実行結果を見てみましょう．

```
push_back
0 a size : 1 capacity : 1
1 a size : 2 capacity : 2
2 a size : 3 capacity : 3
3 a size : 4 capacity : 4
4 a size : 5 capacity : 6
5 a size : 6 capacity : 6
6 a size : 7 capacity : 9
7 a size : 8 capacity : 9
8 a size : 9 capacity : 9
9 a size : 10 capacity : 13
reserve
0 a size : 1 capacity : 10
1 a size : 2 capacity : 10
2 a size : 3 capacity : 10
3 a size : 4 capacity : 10
4 a size : 5 capacity : 10
5 a size : 6 capacity : 10
6 a size : 7 capacity : 10
7 a size : 8 capacity : 10
8 a size : 9 capacity : 10
9 a size : 10 capacity : 10
resize
0 a size : 10 capacity : 10
1 a size : 10 capacity : 10
2 a size : 10 capacity : 10
3 a size : 10 capacity : 10
4 a size : 10 capacity : 10
```

```
5 a size : 10 capacity : 10
6 a size : 10 capacity : 10
7 a size : 10 capacity : 10
8 a size : 10 capacity : 10
9 a size : 10 capacity : 10
```

まず，push_back で要素を確保していった場合です．size 関数の結果は，確保した数と一致しています．しかし，capacity 関数の結果は，確保した数より多くなっている場合があります．これは，push_back で要素を追加するたびにメモリ領域を再確保していったのでは，効率が悪いので，ある程度要素数が増えた場合，それにあわせて余分に領域を予約領域として確保していく，という設計になっているからです．また，この設計のために，先程のようなポインタのエラーがでてしまっています．

次に，reserve 関数で手動確保した場合です．reserve 関数は，その名前 (reserve:予備，蓄え) からも分かるように，メモリを予約領域として確保する関数です．10 要素分確保したい場合は，reserve(10) とします．reserve は予約領域を確保しただけなので，実際に値を追加する場合は，push_back する必要があります．もちろん，先に確保しているので，先程のようなポインタのエラーはなくすことができます．

最後に，resize 関数で手動確保した場合です．resize 関数は，実際に値を代入できる領域として確保します．そのため，例えば，自分が作ったクラスのベクタを用意し，それを resize した場合，そのクラスのデフォルトコンストラクタが実行されます．また，vector<int> v; v.resize(10, 0);(int 型 10 要素を確保して 0 で初期化) といったこともできます．

List.2

```
#include <iostream>
#include <vector>
using namespace std;
int main(void)
{
  vector<int> a;
  int i, n;

  cout << "n : ";
  cin >> n;
  cout << "push_back" << endl;
  for(i = 0; i < n; ++i){
    a.push_back(i);
    cout << i << " a size : " << (int)a.size();
    cout << " capacity : " << (int)a.capacity() << endl;
  }

  a.clear();
  cout << "reserve" << endl;
  a.reserve(n);
  for(i = 0; i < n; ++i){
    a.push_back(i);
    cout << i << " a size : " << (int)a.size();
    cout << " capacity : " << (int)a.capacity() << endl;
  }

  a.clear();
  cout << "resize" << endl;
  a.resize(n);
  for(i = 0; i < n; ++i){
    a[i] = i;
    cout << i << " a size : " << (int)a.size();
    cout << " capacity : " << (int)a.capacity() << endl;
  }

  return 0;
}
```

3.2 値の参照

ベクタ要素の値を直接参照するには，2 つの方法があります．1 つは，配列と同じようにオペレータ [] を用いる方法です．この方法では，配列の場合と同じように範囲外の要素を指定した場合，メモリエラーとなってしまいます．それを回避したい場合は，2 つ目の方法である，`at()` を使います．`at(要素番号)` でアクセスし，要素番号のところにある要素への参照を返します．これは，[] と同じであり，[] を使った方が配列と同じであり，使いやすいのではないかと考えるでしょう．では，なぜ `at()` があるのでしょう．

これは，前述の範囲外要素への参照時の動作に関連します．`at()` による参照では，範囲外の要素にアクセスしようとすると，`out_of_range` 例外が送出されます．List.3 にこれを用いたエラー処理の例を示します．`try-catch` は，`try` 内でエラーが起こった場合，エラー内容を `catch` で受け取り，エラー処理するというものです．

List.3 の実行結果です．

```
error : ベクタ範囲外の要素 10 にアクセス
```

例外が呼び出されていることが分かります．

List.3

```
#include <iostream>
#include <vector>
using namespace std;
int main(void)
{
  vector<int> a;
  int i, n;

  cout << "n : ";
  cin >> n;
  a.resize(n);
  try{
    for(i = 0; i < n+10; ++i){
      a.at(i) = i;
    }
  }
  catch(out_of_range err){
    cout << "errorベクタ範囲外の要素 : " << i;
    cout << "にアクセス" << endl;
  }
  return 0;
}
```

4　コンテナと反復子，アルゴリズム

STL には，**コンテナ**，**反復子**，**アルゴリズム**の 3 つの基本要素があります．(実際には，さらに**アロケータ**，**アダプタ**，**関数オブジェクト**があります．)

4.1　コンテナ

前節までで，vector という動的配列をやってきました．これは，コンテナの一種です．コンテナ (container) とは，他のオブジェクトを格納するオブジェクトです．コンテナには，vector の他に，deque(両端から操作できるキュー)，list(線形リスト) などがあります．

STL の標準的なコンテナを以下に示します．

- vector：配列
- deque：双方向キュー
- list：線形リスト
- queue，stack：キュー、スタック
- map：マップ
- set：集合

それぞれのコンテナについて，ここでは詳しく述べませんので，知りたい場合は各自調べてみてください．

4.2　反復子

反復子 (iterator, イテレータ) は，ポインタに似たもので，コンテナの要素を指します．反復子は，ポインタのように扱え，インクリメント，デクリメントによって，コンテナ内の要素を移動できます．また，逆参照演算子*も適用できます．

では，なぜポインタでなく反復子を用いるのでしょう．例えば、vector は，ポインタを進めれば自然と次の要素を指しますが，list などは次の要素を指すポインタを参照する必要があります．そこで，コンテナの要素を移動する際のインターフェースを統一し，コンテナの種類に依存することなく同様の方法で要素にアクセス出来るようにしたというわけです．

また，反復子はポインタと違い，様々な種類が用意されており，場合によって使い分けることができることです．以下に標準的なものを示します．

- iterator：基本的な反復子
- input_iterator：入力イテレータ
- output_iterator：出力イテレータ
- forward_iterator：前進イテレータ
- bidirectional_iterator：双方向イテレータ
- random_access_iterator：ランダムアクセスイテレータ
- const_iterator：const なコンテナに対するイテレータ

iteratorは，ほとんどの反復子の継承元になっているものです．普段よく使うのは，iteratorやconst_iteratorでしょう．

反復子の使い方の例を List.4 に示します．

List.4

```
#include <iostream>
#include <vector>
using namespace std;
int main(void)
{
  vector<int> a;
  vector<int>::iterator pa;  //反復子
  int i, n;
  cout << "n␣:␣";  cin >> n;
  a.resize(n);

  //値の代入
  pa = a.begin();
  i = 0;
  while(pa != a.end()){
    *pa = i;
    pa++;
    i++;
  }

  //値の出力
  for(pa = a.begin(); pa != a.end(); ++pa){
    cout << *pa << "␣";
  }
  return 0;
}
```

begin() 関数は，コンテナの最初の要素への反復子を返します．また，end() 関数は，コンテナの最後の要素の 1 つ後を示す反復子を返します．最後の要素でなく，その次の要素を返すことで，while(pa != a.end()) のように効率的にコンテナ要素を参照していくことができます (よって，最後の要素は end()-1 となる)．

実行結果を以下に示します．

```
n : 5
0 1 2 3 4
```

4.3 アルゴリズム

アルゴリズム (algorithm) は，コンテナ内の要素を操作します．例えば，要素の初期化，ソート，検索，変換などです．また，多くのアルゴリズムは，コンテナ内のある範囲のみに，これらの操作を実行することができます．

ソート sort の例を List.5 に示します．実行結果は以下となります．

```
n : 10
0 56 19 80 58 47 35 89 82 74
0 19 35 47 56 58 74 80 82 89
```

List.5

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

//乱数を生成
inline int nrand(int n)
{
  return (int)(rand()/(1.0f+RAND_MAX)*n);
}

int main(void)
{
  vector<int> a;              //ベクタ
  vector<int>::iterator pa;  //反復子
  int i, n;
  cout << "n : ";  cin >> n;

  //乱数の代入
  a.resize(n);
  for(i = 0; i < n; ++i){
    a[i] = nrand(100);
  }

  for(pa = a.begin(); pa != a.end(); ++pa){
    cout << *pa << " ";
  }
  cout << endl;

  //ソート
  sort(a.begin(), a.end());

  for(pa = a.begin(); pa != a.end(); ++pa){
    cout << *pa << " ";
  }
  cout << endl;

  return 0;
}
```