

[prog2] Programming C++ (C6) Exercise Guide (Ex02)

10/5, Thursday 3rd period.

Ex02 について

vector... ベクター ⇒ ベクトル？
線形代数の「ベクトル」？
⇒ 複数の値が一方向に並ぶ入れ物


▶ vector

高機能な配列 とでも思えばよい

- ・ 可変長データ配列の操作が可能

[Prog0/C まで]

10個の整数値の配列 ⇒ `int a[10];`

・ 入力のデータ数が `n` 個？ ⇒ `int n;`
 `scanf("%d",&n);`
`int a[n];`

・ 入力数が不明 ⇒ とりあえず、大きめにとる
`#define N 10000`
`int a[N];`

Ex02 について

- ・配列の大きさを、入力の個数に応じて決める

```
int *a;
```

```
a = (int *)malloc(sizeof(int)*n);
```

mallocによる動的確保と解放を使う！

- ・後から配列の要素数を増や(減ら)したくなった

⇒ 授業上は、「無理」(realloc())という関数を使えばできるが…)

⇒ 要素数を任意に変えたいなら、連結リストでも使えば？(実装大変)

めんどくさい

C++では、vectorコンテナを使えば、ずっと楽になるよ！

Ex02 について：STLコンテナ



・**Standard Template Library** (C++の標準装備)

C++において、**データの格納を便利にする**ライブラリ集

《 STLコンテナの種類 》

・**vector**

・**list**

・**map**

・set

・stack

・queue

などなど...

この授業では、代表的な vector, list, mapの3つを取り上げる。

⇒ 使い方が分かれば、基本的な手法はどのコンテナも類似的

・機能や性質、性能を吟味して、**使い分けができるのが理想的**

vector (クラス)を使う

- ・ vector は クラス で定義されているものだが、とりあえず今は深く考えない（後半で、詳しく扱う）⇒ 道具としての利用方法

■ 使い方 ※ <vector> をincludeすること

int 型 のデータ列を持ちたいとき: `vector<int> vec;`

double 型 のデータ列を持ちたいとき: `vector<double> vec;`

`vector<保持する型の名前> 変数名;`

- ・ 先に構造体を定義してあれば、構造体のvectorという構成も可能

ちなみに `using namespace std;`を書かないと、`std::vector<int> ...` と書く必要がある

vector (クラス)を使う

// int型のvectorである”vec”

```
vector<int> vec;
```

```
vec.push_back(2);
```

```
vec.push_back(4);
```

```
vec.push_back(-1);
```

```
vec.push_back(6);
```

“vector名”.push_back(値);
と書くと、値を末尾に追加！
⇒ 配列のサイズはどうなってる？
勝手にやってくれる(追加するたびに大きくなる)から気にしなくてOK!

vectorは大きさ指定不要！
宣言した時点では、
vec は空っぽ！

《vecのデータ》

大きさ

0

2

1

2 4

2

2 4 -1

3

2 4 -1 6

4

vectorの要素にアクセスしたい...

⇒配列感覚でOK!

例)

vec[0] ⇒ 2

vec[1] ⇒ 4

vec[2] ⇒ -1

vec[3] ⇒ 6

vector (クラス)を使う

vector<int> vec; を例に...

■ おさえておきたい機能・基本

v.push_back(x);	// int型の変数 x を vectorに追加
v[要素番号]	//配列風にアクセス可能

typedef vector::size_type vec_sz; //vectorサイズを扱うための準備
vec_sz size = vec.size(); // vector名.size()で vectorのデータ数

sizeだから実質 int型？
⇒ 直接 vec.size() だけでも動くには動く。
丁寧に書くなら2行セットで

例題 1

ストリームからきた入力を vector へ push

```
1  #include <iostream>
2  #include <vector> //vector利用時には、ヘッダーが必要
3
4
5  using namespace std;
6
7  int main(){
8      vector<int> weight; //int型のvectorコンテナ (≒int型の伸縮自在な配列)
9      int w;
10     double sum=0.0;
11     double avg, median;
12
13     cout << "Input values" << endl;
14
15     // cin は 入力失敗 (入力終了時) に 0 を返すので、
16     // 任意の入力個数に対する入力は、このようにwhile文で書ける
17     while(cin >> w){
18         weight.push_back(w); // vectorに入力値 w を末尾に挿入
19     }
20 }
```

while文の条件の中に cin を書ける！
scanfのように入力成功・失敗を管理する戻り値のようなものがある

※ 入力をやめない限り、無限ループするので Ctrl+D で入力を終了すること

例題 1

vectorへのデータを処理

vectorの大きさは伸縮自在。
.size()で要素数はすぐ分かる
⇒全要素を探索するループは簡単

```
21 cout << "[Data]" << endl;
22 for(int i=0;i<weight.size();i++){ // vectorの入力された個数分ループする
23     cout << weight[i] << " ";
24     sum += weight[i]; //合計値計算 (vectorの各要素へは配列と同じようにしてランダムアクセスが行える)
25 }
26 cout << endl;
27
28 //平均値計算
29 avg= sum / weight.size();
30
31 //中央値計算
32 sort(weight.begin(),weight.end());
33 if(weight.size()%2==1){//奇数個の時は真ん中の値
34     median = weight[weight.size()/2];
35 }
36 else{//偶数個のときは、真ん中に近い2つの平均値
37     median = (weight[weight.size()/2-1]+weight[weight.size()/2])/2.0;
38 }
39
40 cout << "Number of Data: " << weight.size() << endl
41     << "total: " << sum << " average: " << avg
42     << " median: " << median << endl;
43
44 return 0;
45 }
```

vector の要素へのアクセス
⇒ 配列と同じ

入力ストリームから異なるデータ列の入力

▶ cin から データ列Aを入力

▶ cin から データ列Bを入力

一見難しくなさそう(同じような処理を2回書けばいい)だが...

データ列の入力をCtrl+Dで打ち切ると、打ち切り(入力失敗)情報が cin に溜まってしまう ⇒ それ以降、入力がずっと失敗する

▶ (別なデータ列への)新しい入力を続ける場合は

`cin.clear();`

を使って、入力失敗情報を消そう

《a,bというvectorに別々に入力》

```
while(cin >> w){  
    a.push_back(w);  
}
```

`cin.clear();`

```
while(cin >> w){  
    b.push_back(w);  
}
```

イテレータを使ってコンテナの操作をしよう

イテレータ ⇒ ポインタのようなもの

STLコンテナの特定位置をアドレスを使って表す。

例えば、`vector<int> vec;` に対するイテレータは次のように宣言する。

```
vector<int>::iterator it;           // int *p; みたいな
```

《イテレータ操作関係》

`vec.begin();` // “vec”という名前のvectorの先頭の位置

`vec.end();` // “vec”という名前のvectorの末尾の位置

- ▶ ポインタを使うのと同じように、***it** とすることで it が指すvectorの要素を間接的に参照できる

例題 2

乱数で vector を初期化 ⇒ イテレータを用いて参照

```
1  #include <iostream>
2  #include <random> //c++用乱数
3  #include <algorithm>
4  #include <vector>
5
6
7  using namespace std;
8
9  int main(){
10     random_device rnd;    // c++用の乱数。とりあえず無視してよい
11     vector<int> vec;
12     int a;
13
14     for(int i=0;i<10;i++){
15         a = rnd()%9+1;    //乱数0~9生成
16         vec.push_back(a);
17     }
18
19
20     //使っているvectorはvector<int>なので、vector<int>に対するイテレータを宣言
21     vector<int>::iterator it,it2;
22     cout << "[Before]" << endl;
23     for(it=vec.begin();it!=vec.end();it++){ // .begin()は先頭アドレス、.end()は末端アドレス
24         cout << *it << " "; //イテレータはポインタと同等なもの。実体を得るためには*をつける
25     }
26     cout << endl;
```

「vector の先頭から末尾まで」というループの書き方。
ポインタと同じスタイルなので、参照先のインクリメントは **it++**; で OK



例題 2

sort()関数(※要<algorithm>のinclude)
対象範囲(先頭、末尾)を示すイテレータを引数に渡すことで、
指定範囲の値をソートしてくれる標準ライブラリ関数

「イテレータ」が分かると可能になる、あれこれ

```
29 //イテレータを引数とするライブラリ関数 (vector自体とは関係ない)
30 sort(vec.begin(),vec.end()); // 第1引数で示されたイテレータから、第2引数で示されたイテレータの直前までをソート
31
32 cout << "[After sort]" << endl;
33 for(it=vec.begin();it!=vec.end();it++){
34     cout << *it << " ";
35 }
36 cout << endl;
37
38 it2 = find(vec.begin(),vec.end(),5); // '5'が見つかるvecの先頭アドレス (イテレータ) を返す
39
40
41 cout << "[Later than first '5' ]" << endl;
42 for(it=it2;it!=vec.end();it++){ // it2 には ソート後のvec内の5が初めて出現した場所が書き込まれている
43     cout << *it << " ";
44 }
45 cout << endl;
46
47
48 return 0;
49 }
```

“vec”内において、5という要素が最初に出現した
位置(it2) よりも後ろの部分の値だけを順に表示

find()関数(※要<algorithm>のinclude)

対象範囲(先頭、末尾)を示すイテレータを第1,2引数に渡すことで、
第3引数の値を検索してくれる標準ライブラリ関数。
⇒ 見つかったら、その場所を指すイテレータを返す。
(見つからなかったら、vec.end()と同じものが返る)

別なサンプル：入力、ソート、削除

vector サンプル（ストリームからきた入力を vector へ push）

```
1  #include <iostream>
2  #include <vector>    //vector用
3  #include <algorithm> //sort用
4
5  using namespace std;
6
7  int main(){
8      int val;//入力値
9      int i;
10     vector<int> vec; //int型の値を持つvector
11     typedef vector<int>::size_type vec_sz;//vectorサイズを持つ型の簡便化
12
13     cout << "[Vector Insert Demo]" << endl;
14     /* vector へ値を挿入するデモ */
15     while(cin >> val){ //入力がある限り受け続ける
16         vec.push_back(val);//vectorへpush
17
18         vec_sz vsize = vec.size(); //vecの現在のサイズ
19         cout << "[vector (size:" << vsize << ")] ";
20         for(i=0;i<vsize;i++){ //vecの中身を出力（配列と同じ書き方でOK）
21             cout << vec[i] << " ";
22         }
23         cout << endl;
24     }
25     cout << endl;
26     cout << "[Vector After sorted...]" << endl;
27 }
```

```
kono@241E-PC ~/cpp
$ ./a.exe
[Vector Insert Demo]
2
[vector (size:1)] 2
5
[vector (size:2)] 2 5
3
[vector (size:3)] 2 5 3
7
[vector (size:4)] 2 5 3 7
9
[vector (size:5)] 2 5 3 7 9
■
```

※ 入力をやめない限り、無限ループするので Ctrl+D で入力を終了すること

別なサンプル：入力、ソート、削除

vector サンプル（vector の要素をソートする）

```
28
29  /* vector をソートして、イテレータを用いて参照・結果を出力するデモ */
30  vec_sz vsize = vec.size();
31  sort(vec.begin(),vec.end()); //ソート (begin()やend()はイテレータ (ポインタみたいなもの) を返す)
32  cout << "[vector (size:" << vsize << ")] ";
33
34  vector<int>::iterator itr;//イテレータによる、ポインタ風の配列アクセス
35  for(itr = vec.begin(); itr!=vec.end(); itr++){ //やっていることは20行目のループと同じ
36      cout << *itr << " "; //ポインタと同じように、中身を参照したい場合は * をつける
37  }
38  cout << endl << endl;
39  cout << "[Vector Delete Demo]" << endl;
```

sort(vec.begin(). vec.end()); で **vector**全体のソート

★ アドレス指定のような形 ⇒ イテレータを使って任意の範囲だけのソートも可能

[vector (size:5)] 2 5 3 7 9

[Vector After sorted...]

[vector (size:5)] 2 3 5 7 9

別なサンプル：入力、ソート、削除

vector サンプル（vector の末尾の要素を取得 or 削除）

```
/* vectorの末尾から値を取り出して、削除するデモ（講義範囲外） */
while(vec.size() > 0){//vectorのサイズが空になるまで
    int last = vec.back();//末尾の値を参照する（この部分は講義範囲外）
    vec.pop_back();//末尾の値を削除する
    cout << "<delete> " << last << " (vec[" << vec.size() << "]) " << "was deleted." << endl;
    cout << "[vector (size:" << vsize << ")] ";

    for(itr = vec.begin(); itr!=vec.end(); itr++){ //残っている要素を出力（イテレータ版）
        cout << *itr << " ";
    }
    cout << endl;
}
```

vec.push_back(x);と真逆の操作

要素数(=vec.size())が1ずつ減る

⇒ メモリ領域も自動的に解放。

freeいらず

[Vector Delete Demo]

<delete> 9 (vec[4]) was deleted.

[vector (size:5)] 2 3 5 7

<delete> 7 (vec[3]) was deleted.

[vector (size:5)] 2 3 5

<delete> 5 (vec[2]) was deleted.

[vector (size:5)] 2 3

<delete> 3 (vec[1]) was deleted.

[vector (size:5)] 2

<delete> 2 (vec[0]) was deleted.

[vector (size:5)]