[prog2] Programming C++ (C6) Exercise Guide (Ex11)

11/13, Monday 3rd period.

Ex11 について

講義の説明の通りに、vectorコンテナを自分で実装（Vec class)

- 教科書に答え自体は全部書いてあります
- ただし、vectorそのものの使いやすさに反比例するかのように
  詳細は難しく複雑な部分もあります

- 「内部実装の複雑さを隠蔽して、ユーザーには使い勝手のよさの
  部分だけを提供している」ということの理解

vectorクラスの自作の前に…

- vectorコンテナを使って何をしていたか？（主なもの）

インターフェースとして (public)：

- ほぼ無制限に値を追加格納できる ・・・ push_back()
- コンテナのサイズを取得する ・・・ size()
- コンテナの指定要素を削除、または全部を削除 ・・・ erase(), clear()

- 格納する変数の型にとらわれない ・・・ vector<T>
- ランダムアクセス（ポインタ、イテレータによる要素間の往来）が可能

・・・ vector<T>::iterator
begin() , end()

内部構造として (ユーザーには見せないprivate)：

- 任意のデータ保存領域を持つ（メモリの動的確保・解放）

⇒ コンテナの大きさをその都度、自由自在に変えられるように

vectorに相当するクラス実装の実態（メモリ）

- vectorに値を挿入、あるいは削除する時

```
vector<int> v;
v.pushback(1);          v.erase(v.begin());
```

などと、簡単にできるが…

- 裏（クラスのprivate部分）では涙ぐましい努力が

ユーザーに呼び出されたpublicな関数に応じて、
メモリ領域の動的確保や解放に関する処理を暗に実行

vectorに相当するクラス実装の実態（メモリ）

- ~~メモリ領域確保なんて、適当に new とか delete しとけば~~

- vectorのいいところ

⇒ 格納する変数などのデータ型にとらわれない実装

int でも　構造体でも クラス、さらにはvectorでも可

⇒　それが逆に裏目に　「自作」の難易度が上がる

new や　delete　は、実行時にコンストラクタ、デストラクタの呼び出しを伴うという性質がある

⇒ **自作の型（構造体）**など、コンストラクタ・デストラクタが実装されていないものに対応できなくなる

原始的（もっと低レベルな）メモリ確保

- 標準ライブラリ <memory> にある allocatorクラス を使う

```
 8  int *data;                  //データを格納する動的配列の先頭アドレス
 9  int *lim;                 //動的配列の末端（厳密には、最終要素の１つ後ろ）のアドレス
10  allocator<int> alloc;     //メモリの動的確保・解放に関するクラスのオブジェクトallocを生成
11  int size;                   //確保するデータサイズ
```

- allocator<T> と、テンプレートの形で定義されているので、真に任意の変数型のメモリ領域を確保可能

⇒ malloc, new のような今までのやり方より数段上の手間

```
20    /* メモリ領域の動的確保 */
21
22    data = alloc.allocate(size);
23    lim = data + size;
24    uninitialized_fill(data,lim,0);
```

```
31  //確保した領域に値を代入、出力するテスト
32  for(int i=0; i<size;i++) data[i]=i;
33  for(int i=0; i<size;i++) cout << data[i]*2 <<endl;
```

```
38  /* 動的に確保したメモリ領域の解放 */
39
40  if(data){
41    int *itr = lim;
42    while(itr != data){
43      alloc.destroy(--itr);
44    }
45    alloc.deallocate(data, lim-data);
46  }
```

自作「Vecクラス」のメモリ確保の方向性

- **教科書を読むと出てくる privateなメンバー関数 grow()**
- ユーザーからコンテナに対して値の挿入を求められたときに、必要なメモリ領域を追加で確保していく裏方の関数。

```
void grow() {
  size_type new_size = max(2 * (limit - data), ptrdiff_t(1));
  iterator new_data = alloc.allocate(new_size);
  iterator new_avail = uninitialized_copy(data, avail, new_data);
  uncreate();
  data = new_data;
  avail = new_avail;
  limit = data + new_size;
  //  cout << "newsize:" << new_size << endl;
}
```

先（これからまだまだ追加されてくるだろうデータに対するメモリ領域増設）を見越して、2のべき乗単位で確保する

⇒ 頻繁に行われるメモリ確保そのものによるパフォーマンス（実行速度など）の低下を軽減するため

⇒ 毎回1個ずつ確保より、ある程度まとめて確保した方が「データ数が非常に大きくなるような」場合に効率的（**メモリ確保の前準備を減らす**）になる

⇒ データ領域の大きさと、実際に値が入っているデータ領域を別個に管理

テンプレートクラス

- Ex08ではテンプレート関数を扱った

⇒ 関数の引数や戻り値を任意の型（抽象的）にする

例） template <typename T> T function(T val);

- テンプレートを使うと、クラスのメンバーの型を抽象的にすることができる（⇒抽象化クラス）

例）
```
template <class X> class Test{
    private:
    X data1;
    X data2;
 public:
    X func ( X a);
};
```

オブジェクト生成：　Test<int> object;