Programming C++
Lecture Note 11
抽象データ型を定義する

Jie Huang

抽象データ型とは

- 抽象データ型は型の定義と手続きの定義からなる実装部分と、手続きの使い方（手続きの引数の数と型）を指定するインターフェース部分に分かれる。
  ⇒実装部分は保護されて、外部からは参照できない。
- 抽象データを使うプログラムはインターフェース部分のみに従って書かれる。
  ⇒インターフェースによって、データ型が完全に決められるので、データ型の抽象と呼ぶ。

vectorクラスの使い方を考える

- 例えばvectorクラスの場合は、以下の操作ができる機能を提供することが仕様によって決められ、ユーザはその仕様に従ってクラスの機能を使うことができる。

```
vector<Student_info> vs; //要素の型をテンプレートによる定義
vector<double> v(100); //要素数の指定
vector<Student_info>::const_iterator b, e; //イテレータ
b = vs.begin(); e = vs.end();
vector<Student_info>::size_type i = 0; //インデックス指定
for (i = 0; i != vs.size(); ++i) cout << vs[i].name();
```

- 以下では、このvectorクラスと同じ機能のクラスを実装することによって、vectorのインターフェースの仕組みを理解する。

新しいvectorクラス、Vecクラスの実装

- Vecにいろんな型のデータを格納するためには、テンプレートを使って定義する。
- 内部構造を実現するのに、配列を使うことを考える。
- 反復子はデータへのポインタを使うことで実現できる。

```
template <class T> class Vec {
public:
  // ここでインターフェースを定義する
private:
  T* data;     // データの先頭を表す反復子
  T* limit;    // 最後のデータの1つ後を表す反復子
};
```

⇒データの先頭と1つ後ろを表す反復子はインターフェースの部分ではないので、外部からアクセスできないように、privateの部分に入れる。

新しいvectorクラス、Vecクラスの実装

- 以下の使い方を実現するためには
```
Vec<Student_info> vs;
Vec<Student_info> vs(100);
```
- コンストラクタの実装
```
template <class T> class Vec {
public:
  Vec() { create(); } // デフォルトコンストラクタ
explicit Vec(size_type n, const T& val = T())
  { create(n, val); }//パラメータコンストラクタ
private: T* data; T* limit;
};
```
- Createはメモリの確保を行う関数
- explicit宣言により、このコンストラクタは型変換には利用されなくなり、以下の代入時におけるコンストラクタの使用を禁ずる。
```
Vec<int> vi = 100; // 誤り
Vec<int> vi(100);  // 正しい
```

必要な型の定義

```
template <class T> class Vec {
 public:
   typedef T* iterator;                  // 型の定義
   typedef const T* const_iterator; // 型の定義
   typedef size_t size_type;             // 型の定義
   typedef T value_type;                 // 型の定義
   Vec() { create(); }
   explicit Vec(size_type n, const T& val = T()){
     create(n, val);
   }
 private:
   iterator data;     // 新しい型を使った反復子の定義
   iterator limit;    // 新しい型を使った反復子の定義
};
```

⇒ここで、反復子はポインタとして実装される

インデックス演算子の定義

- データのインデックス操作

  ```
  T& operator[](size_type i) {return data[i];}
  ```

  - このインデックス演算子はメンバー関数であるため、オブジェクト自身が暗黙のパラメータの役割を果たす。
  - インデックス操作はインデックス演算子[]を多重定義することによって実現する。
- const型データのインデックス操作

  ```
  const T& operator[](size_type i) const{ return data[i];}
  ```

  ここで、インデックス演算子はパラメータも含めて同じものが2つ定義されている。オブジェクト自身がconstと、そうでないことによって区別できるので、2つの同じ関数の定義を可能にしている。

反復子を戻す関数を定義する

- 反復子を戻す関数を定義

```
iterator begin() { return data; }
const_iterator begin() const { return data; }
iterator end() { return limit; }
const_iterator end() const { return limit; }
```

- データのサイズを返す関数

```
size_type size() const {
   return limit-data;
}
```

メモリ管理以外のVecの主な部分の実装

```
template <class T> class Vec {
public:
  typedef T* iterator; typedef const T* const_iterator;
  typedef size_t size_type; typedef T value_type;
  Vec() { create(); }
  explicit Vec(size_type n, const T& val = T()) {
    create(n, val)
  };
  size_type size() const { return limit-data; }
  T& operator[](size_type i) { return data[i]; }
  const T& operator[](size_type i) const {return data[i];}
  iterator begin() { return data; }
  const_iterator begin() const { return data; }
  iterator end() { return limit; }
  const_iterator end() const { return limit; }
private:
  iterator data; iterator limit;
};
```

newとdeleteを使うメモリ管理の問題点

- new演算子はメモリの確保だけでなく、メモリの初期化も自動的に行うので、余分の処理時間がかかる。
- 特にpush_back関数などによりメモリ領域の拡大が必要となる時に、大きな負担となる。
- new演算子はその型のデフォルトコンストラクタが定義されていることが必要で、汎用性が失われる。
- 柔軟なメモリ管理ツールは言語そのものではなく、標準ライブラリに任せた方が、柔軟性がある。

メモリ管理クラスのインターフェース

- 新たなクラスにおけるメモリ管理メンバー関数の定義

```
template<class T> class allocator {
public:
  T* allocate(size_t); // 初期化なしでメモリ確保
  void deallocate(T*, size_t); // メモリ開放
  void construct(T*, const T&); //オブジェクトの生成
  void destroy(T*); // オブジェクトの破棄
  // ...
};
```

- 付随するいくつかの非メンバー関数
  - 確保されたメモリを特定のT型値で埋める

```
tempalte<class Out, class T>
  void uninitialized_fill(Out, Out, const T&);
```

  - 要素をメモリへコピーする

```
tempalte<class In, class Out>
  Out uninitialized_copy(In, In, Out);
```

allocatorクラスを使ったメモリ管理関数

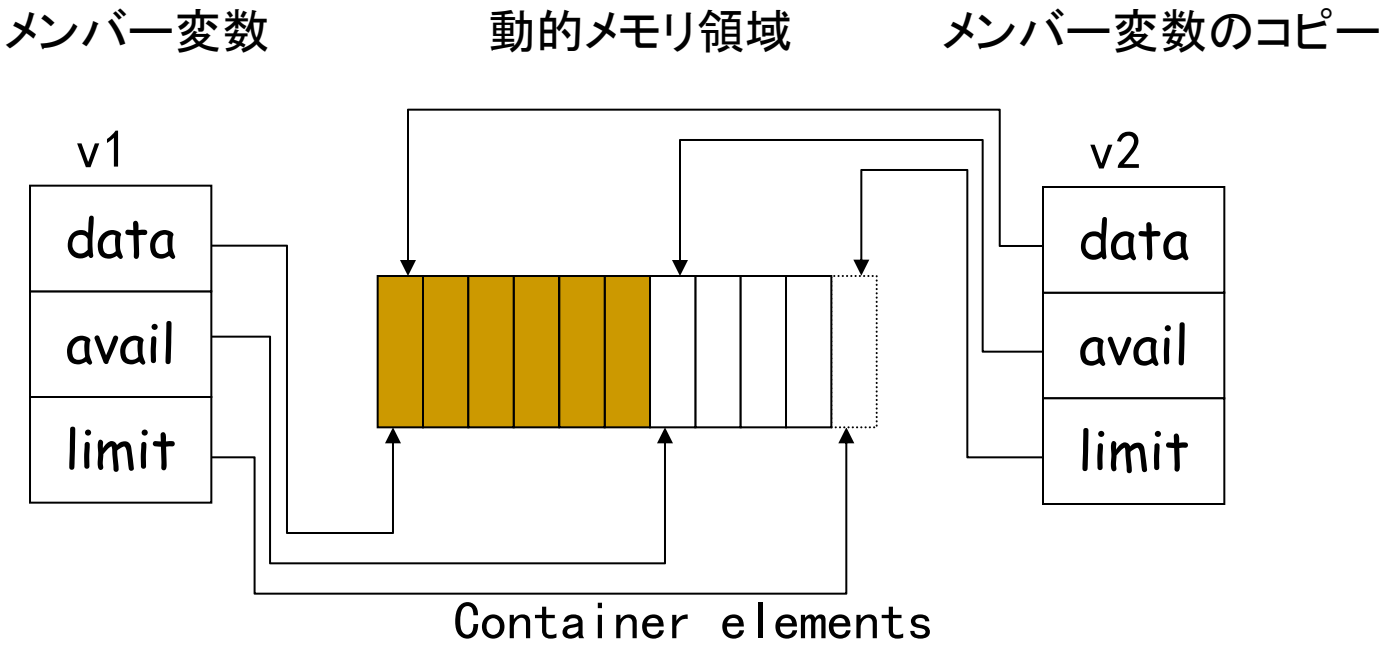
```
private:
    iterator data, avail, limit; std::allocator<T> alloc;
    template <class T>
    void Vec<T>::create(){  data = avail = limit = 0;}
    template <class T>
    void Vec<T>::create(size_type n, const T& val) {
      data = alloc.allocate(n); limit = avail = data + n;
      std::uninitialized_fill(data, limit, val);
    }
    template <class T>
    void Vec<T>::create(const_iterator i, const_iterator j){
      data = alloc.allocate(j - i);
      limit = avail = std::uninitialized_copy(i, j, data);
    }
    template <class T> void Vec<T>::uncreate() {
      if (data) {  iterator it = avail;
        while (it != data) alloc.destroy(--it);
        alloc.deallocate(data, limit - data);
      }
      data = limit = avail = 0;
    }
```

Allocatorクラスと使ったメモリ管理関数

```
// メモリを拡張する関数
template <class T> void Vec<T>::grow() {
  size_type new_size = max(2*(limit - data), ptrdiff_t(1));
  iterator new_data = alloc.allocate(new_size);
  iterator new_avail =
    std::uninitialized_copy(data, avail, new_data);
  uncreate();
  data = new_data; avail = new_avail;
  limit = data + new_size;
}
//チェックなしでデータを追加する関数
template <class T>
void Vec<T>::unchecked_append(const T& val) {
  alloc.construct(avail++, val);
}
// 標準のpush_back関数
void push_back(const T& t){
  if(avail==limit) grow(); unchecked_append(t);
}
```

デフォルトのコピーコンストラクタの不都合

- コピーコンストラクタは新たに定義しなくても、システムによってデフォルトのものが用意されるが、
- デフォルトのコピーコンストラクタはクラスのメンバー変数はコピーするが、
- 動的に確保メモリについてはコピーコピーしない。
- コピーコンストラクタの再定義によって、動的に確保されたメモリも含めてコピーする必要がある。



コピーコンストラクタの定義とその働き

- コピーコンストラクタはコンストラクタの1つ

```
template <class T> class Vec {
public:
  // コピーコンストラクタ
  Vec(const Vec& v){ create(v.begin(), v.end());}
  // 他は前と同じ
};
```

- こんな時にコピーコンストラクタが使われる

```
vector<int> vi;
double d;
d = median(vi); //コピーコンストラクタによる引数のコピー
string line;
vector<Student_info> vs;
vector<Student_info> v2 = vs; // 初期化のためのコピー
vector<string> words = split(line); // 戻り値のコピー
```

代入演算子の定義

```
template <class T> class Vec {
public:
   Vec& operator=(const Vec& v);
   // 他は前と同じ
};
template <class T>
Vec<T>& Vec<T>::operator=(const Vec& rhs) {
   if (&rhs != this) { // 自己代入の場合はなにもしないでよい
     uncreate(); // 左辺の配列を破棄
     create(rhs.begin(), rhs.end()); // 右辺の要素をコピー
  }
   return *this
}
```

⇒ここで、thisはメンバー関数の呼び出しを行ったクラスオブジェクト自身を指すポインタ

3つのコンストラクタの組

- オブジェクト破棄のためのデストラクタ

```
template <class T> class Vec {
public:
  ~Vec() { uncreate(); }
  // 他は前と同じ
};
```

- 3つのコンストラクタの組
  - コピーコンストラクタ、デストラクタ、代入演算子は強く関係している
  - デストラクタが必要なら、一般的にコピーコンストラクタと代入演算子も定義する必要だろう
  - これら3つは1つの組として考えることができる。
- 以下、各種コンストラクタの型をまとめる:

```
T::T();                      //コンストラクタ
T::~T();                     //デストラクタ
T::T(const T&);              //コピーコンストラクタ
T::operator=(const T&); //代入演算子
```