Programming C++
Lecture Note 12
クラスを組み込み型のように扱う

Jie Huang

組み込みオブジェクトの扱い方について

- 演算子を使うことができる。
- 違う型への自動変換ができる。
- 代入したり、コピーしたりすることができる。
- 代入やコピーの際、オリジナルオブジェクトと同じ値を持つが、独立したものである。

- このようなクラスはVecの実装で既にある程度実現したが、ここでは、Strクラスの実装を例に型変換と関連関数を定義し、インタフェースを考えていく。

Vecクラスを利用したStrクラスの定義

```
class Str {
public:
  typedef Vec<char>::size_type size_type;
  Str() { }
  Str(size_type n, char c): data(n, c) { }
  Str(const char* cp) {
    std::copy(cp, cp + std::strlen(cp),
      std::back_inserter(data));
  }
  template<class In> Str(In b, In e) {
    std::copy(b, e, std::back_inserter(data));
  }
private:
  Vec<char> data;
};
```

- 4番目のコンストラクタはテンプレートで定義されているので、いろんなタイプのオブジェクトを使って初期化することができる。
- このクラスは自分で、メモリの動的管理をしていないので、コピーコンストラクタ、代入演算子、デストラクタは必要なく、その部分の仕事はVecクラスが行ってくれる。

型の自動変換

- これまでの実装で実現できた機能
  - オブジェクトのコピー、代入、破棄などについて定義した(Vecクラス利用の部分も含めて)。
  - また、コンストラクタにより、以下のような形のオブジェクトの生成も可能になった。

    ```
    Str s("hello");
    Str t = "hello";
    ```

    ここで、"="という代入演算子は初期化を表すことに注意。
  - 代入演算子

    ```
    Str s;          // sを初期化
    s = “hello”     // sに新しい値を代入
    ```

    Strオブジェクトへの代入は適当なコンストラクタがあれば、自動的に適用されることになる。この場合はconst char*をパラメータとして持つコンストラクタ(explicit宣言されていない)が型変換に利用され、同じ型に変換されてから代入が行われる。

他の演算子の実装

- Strクラスについては、以下の使い方を実現したい

```
cin >> s;  cout << s;      s[i];  s1 + s2;
```

-
```
class Str {
public:
  //インデックス演算子
  char & operator[](size_type i) { return data[i]; }
  const char& operator[](size_type i) const {return data[i];}
  // 他の部分はいままでと同じ
private:
  Vec<char> data;
};
```

- 出力演算子(一般演算子)

```
ostream& operator<<(ostream& os, const Str& s) {
  for (Str::size_type i = 0; i != s.size(); ++i) os << s[i];
  return os;
}
size_type size() const { return data.size(); }
```

他の演算子の実装

- 入力演算子(一般演算子)：最初の試み

```
istream& operator>>(istream& is, Str& s){
  s.data.clear(); char c;
  while (is.get(c) && isspace(c));
  if(is) {
    // dataがprivate部分にあるので、
    // この部分はコンパイルエラーを引き起こす
    do s.data.push_back(c);
    while (is.get(c) && !isspace(c));
    if (is) is.unget(); // 空白をストリームに戻す
  }
  return is;
}
```

- 上のプログラムはコンパイルエラーを引き起こす。
- 問題の解決は、dataをpublic部分に移すことでも解決できるが、クラスの抽象化とデータ保護の主旨と反する
- 以下では、フレンド関数という新しい概念を導入する。

フレンドを使った入力演算子

- フレンド宣言は特定の関数やあるクラスの全関数に自分のクラスのアクセス保護を解除する役割を果たす

- 入力演算子に対してフレンドを宣言する

```
class Str {
  friend std::istream& operator>>(std::istream&, Str&);
  // 他の部分は以前と同じ
}
```

⇒この部分があれば、前述のプログラムは問題なくコンパイルできるようになる。

その他の2項演算子

- 以下の使い方を考える。

```
s = s1 + s2 + s3;
s = s + s1;
s += s1;
```

- まず、+=演算子(メンバー演算子)の実装を考える
  +=演算子は左辺のオペランドを変更する必要があるので、クラスのメンバー関数にした方がよい。

```
class Str {
public:
  Str& operator+=(const Str& s) {
    str::copy(s.data.begin(), s.data.end(),
        std::back_inserter(data));
    return *this;
  }
  // 他の部分は以前と同じ
};
```

その他の2項演算子

- 次に+=演算子を使って+演算子(一般演算子)を定義する
  +演算子は左右どちらのオペランドとも変更しないので、クラスのメンバー関数にする必要は特にない。

```
Str operator+(const Str& s, const Str& t){
  Str r = s;
  r += t; return r;
}
```

複合エクスプレッションと型変換

- 以下のエクスプレッションを考える
  ```
  const Str greeting = "Hello, " + name + "!";
  ```
- +演算子は左結合であるため、以下のように書き直せる。
  ```
  const Str greeting = ("Hello, " + name) + "!";
  ```
- 順に文字列リテラルとStr型の結合、Str型と文字列リテラルとの結合があることがわかる。
- このいずれの場合でもクラスのコンストラクタで型の自動変換が定義されているので、コンパイルが自動的にconst char*型からStr型への自動変換を適用し、Str型同士の結合を行うようになっているので、何ら問題はない。
- 自動型変換は、一般関数の場合は演算子の左右どちらのオペランドに対しても有効であるが、
- クラスのメンバー演算子に対しては右オペランドに対してしか有効ではない。
- 従って、+のような左右対象の演算子は一般演算子の方が一般的で、また、代入演算子や+=演算子などはクラスのメンバー演算子にするのが一般的である。

コンストラクタを型変換に使わないようにする

- Vecクラスのコンストラクタ

```
explicit Vec(size_type n, const T& val = T()){
  create(n, val);
}
```

ここで、size_typeのnはVecの要素数が示す。

  - このコンストラクタはexplicit宣言であるため、
    `Vec<int> vi(100);`
    はできるが、
  - 以下のような代入の形の型変換では使えないことになっている
    `Vec<int> vi = 100;`
    この代入の形はviに100という値を代入すると誤解されやすい。
    このコンストラクタを型変換に使うと誤解を招く恐れがあるので、このような使い方を避けているのである。

明示的な型変換演算子

- 例えば、Student_infoからdouble型への変換を定義する場合はクラスで以下のメンバー演算子を加えることができる。

```
class Student_info {
public:
  operator double();  // 型変換演算子の中身は省略する
  // ...
};
```

- このような型変換を使って例えば以下のような使い方ができる

```
vector<Student_info> vs;
double d = 0;
for (int i = 0; i != vs.size(); ++i)
  d += vs[i]; // vs[i]は自動的にdoubleに型変換される
cout << "成績の平均:"  << d/vs.size() << endl;
```

- 自動型変換の別の例

```
if (cin) { /* ... */ }
```

ここで、cinは0か0でないvoid*の値を戻すので、それがさらに型変換して、boolの型に変換されることになる。

型変換とメモリ管理

- Strクラスもstringクラスもヌル文字を最後に付けた文字列(文字列リテナル)への型変換がほしくなるかもしれない。しかし、ヌル文字を付けた文字列への変換はポインタ型を渡すことになるので、メモリ管理のことが問題になる。
- 例えば以下のような型変換の演算子が定義されているとする

```
class Str {
public:
  // これらの演算子はもっともらしいが、問題がある
  operator char*();
  operator const char*() const;
  // 他は以前と同じ
private:
  Vec<char> data;
};
```

- まず、最初の型変換はデータのカプセル化を壊す恐れがあるので、適当ではないことがわかる。
- 2番目の型変換はもっともに見えるが、しかし、この演算子はポインタを渡すので、オブジェクトが破棄された後にも使われる可能性があり、エラーの元になりかねない。

型変換とメモリ管理

- また、もし、ここで、新しい領域を確保して、そのポインタを渡すとすると、メモリ領域解放の責任が生じることになる。例えば、以下の使い方では、一時的に生じたメモリ領域の解放は不可能となる。

```
Str s;
ifstream is(s);
```

ここで、一時的にヌルで終る文字列を格納する領域が生成されるが、そのポインタは後で消滅されるので、その領域の解放が不可能となる。

- 標準ライブラリにおける文字列リテラルへの変換
  - 自動変換ではなく、c_str関数によって、明示的にstringの内容をコピーし、そのポインタを渡す。
  - 文字列リテラルの内容はstringクラスが保持し、ユーザはその内容をdeleteしないと考える。
  - 文字列リテラルのデータはstringの他の関数によって、変更される可能性があるので、保持したいユーザはコピーして自分で管理する。