Programming C++
Lecture Note 14
ハンドルクラスによるメモリ管理

Jie Huang

ハンドルクラスによるメモリ管理

- ポインタを利用した処理における問題点
  - ポインタのコピーは指しているオブジェクトのコピーにならない
  - ポインタの破棄がオブジェクトの破棄にはならない
  - オブジェクトを破棄して、ポインタを残すと、どこも指していないポインタになってしまう
  - ポインタを初期化しないで利用すると、オブジェクトを指していないことになる
- 一般的なハンドルクラスに対する要求（スマートポインタ）
  - Handleはオブジェクトを指す
  - Handleはオブジェクトをコピーできる
  - Handleオブジェクトが有効な内部データを指しているかをテストできる
  - Handleオブジェクトが派生クラスのオブジェクトを指す場合に多態性が反映される

ハンドルクラスによるメモリ管理

- 一般的なハンドルクラスの設計
  - 前章のハンドルクラスStudent_infoは学生データ操作のインターフェースと、内部のデータへのハンドルという、2つの独立するべき部分を持ち、よいデザインとはいえない。
  - 新しい一般的なハンドルクラスはハンドルの振る舞いをカプセル化したもので、扱っているオブジェクトとは独立し、テンプレートを使って定義する

一般的なハンドルクラスの定義

- インターフェースクラスとは独立で、ポインタの扱いとメモリ管理を行う一般的なハンドルクラスを定義する

```
template <class T> class Handle {
public:
  Handle(): p(0) { }
  Handle(const Handle& s): p(0){if (s.p) p = s.p->clone();}
  Handle& operator=(const Handle&);
  ~Handle() { delete p; }
  Handle(T* t): p(t) { }
  operator bool() const { return p; }
  T& operator*() const;  T* operator->() const;
private:
  T* p;
};
template <class T> Handle<T>& Handle<T>::
operator=(const Handle& rhs) {
  if (&rhs != this) { delete p;
    p = rhs.p ? rhs.p->clone() : 0;
  }
  return *this;
}
```

ハンドルクラスの演算子の定義

- *演算子の定義

```
template <class T> T& Handle<T>::operator*() const {
  if (p)  return *p;
  throw runtime_error( “unbound Handle” );
}
```

*操作はハンドルの持つオブジェクトへ適用したものになる。例えば、*studentは*(student.p)を返す。

ハンドルクラスの演算子の定義

- ->演算子

```
template <class T> T* Handle<T>::operator->() const {
  if (p)  return p;
  throw runtime_error( “unbound Handle” );
}
```

  - 組み込みの->演算子は一般的な2項演算子ではなく、右オペランドが左オペランドのメンバー名である。
  - 一般的に関数でメンバー名を引数として取ることはできないので、クラス定義->演算子を2項演算子にすることができない。
  - 代わりに、->を単項演算子として定義し、その役割は左オペランドをポインタとして扱えるものに替え、->演算子をもう一回適用することにある。従って、その見かけ上の右オペランドは適用結果の左オペランドのメンバでなくてはならない。
  - 例えば、x->y （xは->演算子を定義したクラスオブジェクト）は->演算子が2回（1つはクラス定義の演算子、もう1つは組み込み演算子）働くことになり、(x.operator->())->y、つまり、x.p->yと同じ意味になる。
  - もし、x.pがポインタではなくて、しもかクラス定義の->演算子を持っているとすると、さらに->演算子が適用され、再帰的なプロセスとなる。

一般的なハンドルクラスを直接使う

```
bool compare_Core_handles(const Handle<Core>& lhs, const
Handle<Core>& rhs) {
  return compare(*lhs, *rhs);
}
int main()  {
  vector<Handle<Core>> students;
  Handle<Core> record;
  char ch;  string::size_type maxlen = 0;
  while (cin >> ch) {
    if (ch ==  'U' )  record = new Core;
    else  record = new Grad;
    record->read(cin);
    maxlen = max(maxlen, record->name().size());
    students.push_back(record);
  }
  // `compare' must be rewritten
  // to work on `const Handle<Core>&'
  sort(students.begin(), students.end(),
    compare_Core_handles);
```

一般的なハンドルクラスを直接使う(続く)

```
// write the names and grades
 for (std::vector< Handle<Core> >::size_type i = 0;
      i != students.size(); ++i) {
   // `students[i] ‘ is a `Handle’,
   // which we dereference to call the functions
   cout << students[i]->name()
     << string(maxlen+1-students[i]->name().size(), ‘ ’);
   try {
     double final_grade = students[i]->grade();
     streamsize prec = cout.precision();
     cout << setprecision(3) << final_grade
          << setprecision(prec) << endl;
   } catch (domain_error e) { cout << e.what() << endl; }
     //ハンドルクラスがメモリを管理するのでdeleteは必要ない
 }
 return 0;
}
```

Student_infoをインターフェースとして定義し直す

- ポインタの扱いと動的メモリ管理を扱うハンドルクラスを使って、Student_infoクラスを純粋のインターフェースとして再定義する。

```
class Student_info {
public: Student_info() { }
  Student_info(std::istream& is) { read(is); }
  //動的メモリ管理の部分が不要
  std::istream& read(std::istream&);
  std::string name() const { if (cp) return cp->name();
    else throw std::runtime_error( “uninitialized Student” );
  }
  double grade() const { if (cp) return cp->grade();
    else throw std::runtime_error( “uninitialized Student” );
  }
  static bool compare(const Student_info& s1,
                      const Student_info& s2) {
    return s1.name() < s2.name();
  }
private:  Handle<Core> cp;
};
```

Student_infoをインターフェースとして定義し直す

- 変更に合わせて、read関数を再定義する
  ここで、ポインタの指すオブジェクトの開放などメモリ管理を気にする必要がないことに注意

```
istream& Student_info::read(istream& is){
  // delete cp; この部分も必要ない
  char ch;
  is >> ch;      // get record type
  if (ch ==  'U' )
    cp = new Core(is);
  else
    cp = new Grad(is);
  return is;
}
```

新しいStudent_infoクラスを使う(前章と同じ)

```
int main(){
  vector<Student_info> students; Student_info record;
  string::size_type maxlen = 0;
  while (record.read(cin)) {
    maxlen = max(maxlen, record.name().size());
    students.push_back(record); }
  sort(students.begin(), students.end(), Student_info::compare);
  for (std::vector<Student_info>::size_type i = 0;
    i != students.size(); ++i) {
    cout << students[i].name()
         << string(maxlen + 1 - students[i].name().size(), ' ');
    try {
      double final_grade = students[i].grade();
      streamsize prec = cout.precision();
      cout << setprecision(3) << final_grade
           << setprecision(prec) << endl;
    } catch (domain_error e) { cout << e.what() << endl; }
  }
  return 0;
}
```

参照カウンタ付きハンドル

```
template <class T> class Ref_handle {
public:
  Ref_handle(): p(0), refptr(new size_t(1)) { }
  Ref_handle(T* t):  p(t), refptr(new size_t(1)) { }
  Ref_handle(const Ref_handle& h) : p(h.p),
    refptr(h.refptr) { ++*refptr;}
  Ref_handle& operator=(const Ref_handle&);
  ~Ref_handle();
  operator bool() const { return p; }
  T& operator*() const {
    if (p) return *p;
    throw std::runtime_error( “unbound Ref_handle” );
  }
  T* operator->() const {
    if (p) return p;
    throw std::runtime_error( “unbound Ref_handle” );
  }
private: T* p;  std::size_t* refptr;
};
```

参照カウンタ付きハンドル

```
template <class T>
Ref_handle<T>&
Ref_handle<T>::operator=(const Ref_handle& rhs) {
  ++*rhs.refptr;
  if (--*refptr == 0) { delete refptr; delete p; }
  refptr = rhs.refptr;
  p = rhs.p;
  return *this;
}
template <class T> Ref_handle<T>::~Ref_handle(){
  if (--*refptr == 0) { delete refptr; delete p; }
}
```

このハンドルクラスはコピーオブジェクトを作る時にデータオブジェクトをコピーしないで、ポインタだけを渡し、カウンタをインクリメントする。また、オブジェクトの破棄はカウンタをデクリメントする。デクリメントの結果、カウンタが0になったら、データの破棄を行う。

データ共有を決められるハンドルクラス

- 今までデータオブジェクトをコピーするハンドルとコピーしないでカウンタを増やすハンドルを学んだが、ここでは、コピーするかどうかを決められるハンドルについて学ぶ。ポインタの代わりになるという意味で、Ptrという名前を付けた。

```
template <class T> class Ptr {
public:
  //new member to copy the object conditionally when needed
  void make_unique() {
    if (*refptr != 1) {
      --*refptr;
      refptr = new size_t(1);
      p = p? clone(p): 0; // p->clone(p);ではない理由は後述
    }
  }
  Ptr(): p(0), refptr(new size_t(1)) { }
  Ptr(T* t): p(t), refptr(new size_t(1)) { }
  Ptr(const Ptr& h): p(h.p), refptr(h.refptr){ ++*refptr; }
```

データ共有を決められるハンドルクラス

```
    // implemented analogously to 14.2/261
    Ptr& operator=(const Ptr&);
    // implemented analogously to 14.2/262
    ~Ptr();
    operator bool() const { return p; }
    // implemented analogously to 14.2/261
    T& operator*() const;
    // implemented analogously to 14.2/261
    T* operator->() const;
private: T* p; std::size_t* refptr;
};
template<class T> T* clone(const T* tp){return tp->clone();}
template<class T> T& Ptr<T>::operator*() const {
    if (p) return *p;
    throw std::runtime_error( “unbound Ptr” );
}
```

データ共有を決められるハンドルクラス

```
template<class T>
T* Ptr<T>::operator->() const {
  if (p) return p;
  throw std::runtime_error( “unbound Ptr” );
}
template<class T> Ptr<T>& Ptr<T>::operator=(const Ptr& rhs){
  ++*rhs.refptr;
  // free the lhs, destroying pointers if appropriate
  if (--*refptr == 0) { delete refptr; delete p; }
  // copy in values from the right-hand side
  refptr = rhs.refptr;
  p = rhs.p;
  return *this;
}
template<class T> Ptr<T>::~Ptr(){
  if (--*refptr == 0) { delete refptr; delete p; }
}
```

Ptrクラスの汎用性

- 例えばStrクラスをPtrを使って定義し直してみる。make_unique関数がp->clone()関数を呼び出しているとすると、メンバー関数としてcloneが必要となる。しかし、Vecクラスにはそんな関数はない。従って、cloneは一般関数として定義する必要がある。
- ```
  template<> Vec<char>* clone(const Vec<char>*);
  class Str {
    friend std::istream& operator>>(std::istream&, Str&);
    friend std::istream& getline(std::istream&, Str&);
  public:
    Str& operator+=(const Str& s) {
      data.make_unique();
      std::copy(s.data->begin(), s.data->end(),
                std::back_inserter(*data));
      return *this;
    }
    typedef Vec<char>::size_type size_type;
    Str(): data(new Vec<char>) { }
    Str(const char* cp): data(new Vec<char>)  {
      std::copy(cp, cp + std::strlen(cp),
        std::back_inserter(*data));
   }
  ```
  ```

Ptrクラスの汎用性

```
    Str(size_type n, char c): data(new Vec<char>(n, c)) { }
    template <class In> Str(In i, In j):data(new Vec<char>){
      std::copy(i, j, std::back_inserter(*data));
    }
    char& operator[](size_type i) { data.make_unique();
      return (*data)[i]; }
    const char& operator[](size_type i) const
      {return (*data)[i];}
    size_type size() const { return data->size(); }
    typedef char* iterator; typedef const char* const_iterator;
    iterator begin() { return data->begin(); }
    const_iterator begin() const { return data->begin(); }
    iterator end() { return data->end(); }
    const_iterator end() const { return data->end(); }
  private: Ptr< Vec<char> > data;
  };
  std::ostream& operator<<(std::ostream&, const Str&);
  Str operator+(const Str&, const Str&);
```

テンプレート特化関数による汎用性

- Ptrクラスにおいて、一般関数のcloneが使われる。それを以下の様に定義する。

```
template <class T> T* clone(const T* tp){
  return tp->clone();
}
```

- この関数は間接的にオブジェクトのclone関数を呼び出すが、オブジェクトがcloneメンバー関数を定義していない場合は汎用性が失われるので、以下のテンプレート特化関数により、特別のバージョンのclone関数を定義することができる。

```
template<> Vec<char>* clone(const Vec<char>* vp){
  return new Vec<char>(*vp);
}
```

この関数は特定の引数型(この場合はVec<char>*)に対して、特定バージョンのテンプレート関数を定義できる。テンプレートの引数にVec<char>*型引数が渡されると、コンパイルは存在しないVecクラスのclone関数を呼び出す代わりに、上記の特化されたclone関数を呼び出すことになる。